The Architecture of Open Source Applications

The Architecture of Open Source Applications

Elegance, Evolution, and a Few Fearless Hacks

Edited by Amy Brown & Greg Wilson

オープンソースアプリケーションのアーキテクチャ
編集:Amy Brown and Greg Wilson
翻訳:Arai Kunimitsu and TAKAGI Masahiro

本書の全文は、http://www.aosabook.org/ でオンラインで読めます。
(英語版の) 印税はすべて、アムネスティ・インターナショナルに寄付されます。

Revision Date: 平成 26 年 11 月 14 日

我々にすべてを教えてくれた Brian Kernighan に
そして世界中にいる良心の囚人たちに

目次

導入

Amy Brown & Greg Wilson

大工仕事は非常に奥の深いものであり、人はみな、上達するための方法を一生涯かけて学び続けることになる。しかし、大工仕事と建築様式は異なる。ピッチ板や留め継ぎの世界から一歩離れて見渡せば、建造物全体を見た設計が必要になる。そしてそれは、技術的・科学的であるのと同程度に芸術の要素もある。

プログラミングもまた奥の深い作業であり、人はみな、上達するための方法を一生涯かけて学び続けることになる。しかし、プログラミングとソフトウェアアーキテクチャは異なる。多くのプログラマは、何年もかけて大規模な設計の問題に取り組む。「このアプリケーションを拡張可能にすべきだろうか？」「仮にそうだとして、その手法はどうする？スクリプトで拡張できるようにするのかプラグイン的な仕組みを取り入れるのか、あるいはまったく異なる別の方法を考える？」「クライアント側でやるべき処理とサーバー側でやるべき処理の切り分けはどうする？そもそもこのアプリケーションを“クライアント・サーバー”型で考えるのは適切なのか？」といった問題だ。これらの問いは、プログラミングに関するものではない。「階段をどこに配置するか」という問いが大工仕事とは関係ないのと同じことだ。

建築様式とソフトウェアアーキテクチャには共通点も多いが、決定的な違いがひとつある。建築家はその生涯を通じて何千ものビルについて研究を重ねるが、大半のソフトウェア開発者はほんの一握りの大規模ソフトウェアしか知ることがない。しかも、その数少ないソフトウェアは自分たちが書いたものであることが多い。ソフトウェア開発者は歴史上の偉大なプログラムを振り返ることもないし、そういったプログラムの設計に関する熟練者の批評を読むこともない。その結果、先人の成功例を参考にすることもできずに同じ過ちを繰り返す。

そんな状況をどうにかしたいと思って本書を書いた。各章では、オープンソースアプリケーションのアーキテクチャについて解説している。どのような構造になっているのか、各パーツがどのように絡み合っているのか、なぜその方式を採用したのか、他の設計上の問題に適用できそうな教訓は何か、といった内容だ。執筆者はそのソフトウェアをもっともよく知る人たちで、何年あるいは何十年もの間、複雑なアプリケーションの設計を経験してきた。本書ではさまざまなアプリケーションを取り上げる。シンプルなドローツールやウェブベースの表計算ソフトもあれば、コンパイラツールキットや数百万行規模の視覚化パッケージもある。数年前に生まれたばかりのアプリケーションもあれば、30 周年を迎えるアプリケーショ

ンもある。すべてのアプリケーションに共通しているのは、作者が長い時間をかけて真剣に設計を考えたこと。そしてその考えを皆で分かち合いたいと考えているということだ。きっと読者のみなさんにも楽しんでもらえるだろう。

執筆者

Eric P. Allman (Sendmail): Eric Allman は sendmail や syslog そして trek の原作者であり、Sendmail, Inc. の共同創業者でもある。彼がオープンソースソフトウェアを書き始めたころにはまだ “オープンソース” などという名前はついておらず、ましてや今のような “ブーム” にはなっていなかった。彼は *ACM Queue* Editorial Review Board および Cal Performances Board of Trustees のメンバーである。個人サイトは `http://www.neophilic.com/~eric` だ。

Keith Bostic (Berkeley DB): Keith はカリフォルニア大学バークレー校の Computer Systems Research Group のメンバーだった。そこで 2.10BSD リリースのアーキテクトや 4.4BSD および関連リリースの開発リーダーをつとめた。彼は USENIX Lifetime Achievement Award (“The Flame”) を受賞した。これは Unix コミュニティへの並はずれた貢献を認められたものだ。また、カリフォルニア大学バークレー校から Distinguished Achievement Award も受賞している。これは 4BSD リリースをオープンソースにしたことに対するものだ。Keith は Berkeley DB のアーキテクトかつ開発者の一員だった。Berkeley DB は、オープンソースの組み込みデータベースシステムである。

*Amy Brown (**編集担当**)*: Amy はウォータールー大学で数学の学士号を取得し、ソフトウェア業界で 10 年の勤務経験を持つ。現在は、書籍の執筆や編集に携わりつつ時にはソフトウェアも書く。彼女は歌手でもあり、他の人のライブを仕切ったりもする - プロもいれば趣味の人もいる。

C. Titus Brown (Continuous Integration): Titus は、進化的モデリングや物理気象学、発生生物学、ゲノミクス、そしてバイオインフォマティクスを研究している。現在はミシガン州立大学の准教授であり、科学的ソフトウェアの再現性や保守性にまで興味の範囲を広げている。彼は Python Software Foundation のメンバーでもあり、ブログは `http://ivory.idyll.org` にある。

Roy Bryant (Snowflock): ソフトウェアアーキテクトおよび CTo として 20 年の経験を持つ Roy は、Electronics Workbench(現在の National Instruments’ Multisim) や Linkwalker Data Pipeline といったシステムを設計した。Linkwalker Data Pipeline は、Microsoft’s worldwide Winning Customer Award for High-Performance Computing を 2006 年に受賞した。最後に在籍したスタートアップを売却した彼はトロント大学に戻り、大学院でコンピュータサイエンスを研究している。専門は、ビジュアライゼーションとクラウドコンピューティングだ。最近は、ACM の Eurosys Conference in 2011 で Snowflock 用の Kaleidoscope 拡張について発表した。個人サイトは `http://www.roybryant.net/` である。

Russell Bryant (Asterisk): Russell は Digium, Inc. のオープンソースソフトウェアチームでエンジニアリングマネージャーを務めている。また、2004 年の秋から Asterisk 開発チームのコ

アメンバーとして活動している。これまでに、Asterisk の開発におけるほぼすべての分野に貢献をしてきた。プロジェクトの運営からアーキテクチャ設計、そして開発まで。彼のブログは `http://www.russellbryant.net` である。

Rosangela Canino-Koning (Continuous Integration): ソフトウェア業界の最前線での 13 年間を経て Rosangela は大学に戻り、ミシガン州立大学でコンピュータサイエンスと進化生物学の博士号取得を目指している。空き時間には読書やハイキング、旅行などを楽しむほか、オープンソースのバイオインフォマティクスソフトウェアをハックすることもある。彼女のブログは `http://www.voidptr.net` である。

Francesco Cesarini (Riak): Francesco Cesarini が Erlang を常用しはじめたのは 1995 年のことだった。その後も Ericsson でさまざまなプロジェクトに参加し、OTP R1 リリースにもかかわっている。彼は Erlang Solutions の創設者であり、O'Reilly の *Erlang Programming* の共著者でもある。現在は Erlang Solutions のテクニカルディレクターとして働いているが、イギリスのオックスフォード大学やスウェーデンのヨーテボリ大学で学生や院生を教えることもある。

Robert Chansler (HDFS): Robert は Yahoo!に在籍するソフトウェア開発のシニアマネージャーである。カーネギーメロン大学の大学院で分散システムを研究した彼はその後、コンパイラ (Tartan Labs)、印刷・画像処理システム (Adobe Systems)、電子商取引 (Adobe Systems, Impresse)、SAN 管理 (SanNavigator, McDATA) などにかかわった。分散システムや HDFS の世界に戻ってきた彼は、解決すべき課題が以前とあまり変わっていないことに気付いた。しかし、登場する数値はどれもみな、ゼロが 2 つか 3 つ多くなっていた。

James Crook (Audacity): James はアイルランドのダブリンに住むソフトウェア開発者。現在は電子工学設計用のツールにかかわっているが、かつてはバイオインフォマティクスソフトウェアを開発していたこともある。彼は Audacity に関する多くの野望を抱えており、その中のいくつかだけでも日の目を見ることを望んでいる。

Chris Davis (Graphite): Chris はソフトウェアコンサルタントであり、Google のエンジニアとしてスケーラブルな監視・自動化ツールの設計と構築に 12 年以上携わっている。Chris が Graphite を書き始めたのは 2006 年で、それ以降ずっとこのプロジェクトを率いている。コードを書いていないときの彼は、料理や作曲そして研究などをしている。彼の研究分野は、知識モデリングや群論、情報理論、カオス理論、そして複雑系などだ。

Juliana Freire (VisTrails): Juliana は、ユタ大学のコンピュータサイエンスの准教授である。それ以前には、ベル研究所 (ルーセント・テクノロジーズ) のデータベースシステム研究部門に在籍したりオレゴン健康科学大学/オレゴン科学技術大学院大学に準教授として在籍したりしていた。彼女の研究分野は、起源や科学データ管理、情報統合、そしてウェブマイニングなどだ。彼女は NSF CAREER および IBM Faculty Award を受賞している。また、彼女の研究に対して国立科学財団やエネルギー省、国立衛生研究所、そして IBM や Microsoft、Yahoo! が資金提供している。

Berk Geveci (VTK): Berk は、Kitware で科学計算のリーダーを務めている。彼は ParaView の開発リーダーでもある。ParaView は、VTK をベースとした視覚化アプリケーションであ

る。彼の研究分野は、大規模なパラレルコンピューティングや計算力学、有限要素、そして視覚化アルゴリズムだ。

Andy Gross (Riak): Andy Gross は Basho Technologies のアーキテクト長であり、Basho のオープンソースおよびエンタープライズデータストレージシステムの設計と開発を仕切っている。Andy が Basho を立ち上げたのは 2007 年 12 月。10 年におよぶソフトウェア開発や分散システムエンジニアリングの経験を経た後のことだった。Basho の前に Andy は、分散システムエンジニアリングの上級技術者として Mochi Media や Apple, Inc.、Akamai Technologies などに勤めていた。

Bill Hoffman (CMake): Bill は Kitware, Inc. の CTO を務める共同創業者である。彼は CMake プロジェクトの主要な開発者であり、大規模な C++システムに 20 年以上携わってきた経験を持つ。

Cay Horstmann (Violet): Cay はサンノゼ州立大学でコンピュータサイエンスの教授を務めるが、しょっちゅう休暇をとっては業界で働いていたり外国で教えていたりする。プログラミング言語やソフトウェア設計に関する多くの著作があり、オープンソースの Violet や GridWorld の原作者でもある。

Emil Ivov (Jitsi): Emil は Jitsi プロジェクト (かつては SIP Communicator と呼ばれていた) の創設者であり、プロジェクトを率いている。彼は、ice4j.org や JAIN SIP プロジェクトなど他の場所でも活躍している。Emil は 2008 年初めにストラスブール大学で博士号を取得した。それ以降は、Jitsi 関連の活動に重点を置いている。

David Koop (VisTrails): David はユタ大学のコンピュータサイエンスの博士候補 (2011 年夏に修了予定)。彼の研究分野は、視覚化や起源そして科学データ管理だ。彼は VisTrails システムのリード開発者であり、VisTrails, Inc. の上級ソフトウェアアーキテクトである。

Hairong Kuang (HDFS) は、貢献者およびコミッターとして長期にわたって Hadoop プロジェクトに長年かかわってきた。かつては Yahoo!で、そして現在は Facebook で働いている。業界で働くようになる前は、彼女はカリフォルニア州立工科大学ポモナ校の准教授だった。カリフォルニア大学アーバイン校でコンピュータサイエンスの博士号を取得している。彼女の研究分野は、クラウドコンピューティングやモバイルエージェント、パラレルコンピューティング、そして分散システムである。

H. Andrés Lagar-Cavilla (Snowflock): Andrés はソフトウェアシステムの研究者で、視覚化やオペレーティングシステム、セキュリティ、クラスタコンピューティング、モバイルコンピューティングなどを対象としている。学士号はアルゼンチンで、そしてコンピュータサイエンスの修士号と博士号はトロント大学で取得した。オンラインでは `http://lagarcavilla.org` で活動している。

Chris Lattner (LLVM): Chris はソフトウェア開発者で、幅広い分野の経験を持つ。コンパイラツール群やオペレーティングシステム、そしてグラフィックや画像レンダリングが得意分野だ。彼は、オープンソースの LLVM プロジェクトの設計者でありリードアーキテクトである。Chris や彼のプロジェクトに関する詳細な情報は `http://nondot.org/~sabre/` で得られる。

Alan Laudicina (Thousand Parsec): Alan はウェイン州立大学の修士課程の学生で、分散コン

ピューティングを学んでいる。空き時間には、コードを書いたりプログラミング言語を学んだり、あるいはポーカーをプレイしたりする。詳細な情報は http://alanp.ca/ で得られる。

Danielle Madeley (Telepathy): Danielle はオーストラリアのソフトウェアエンジニアで、Collabora Ltd. で Telepathy その他の開発にかかわっている。彼女は電子工学とコンピュータサイエンスの学士号を持っており、Plush Penguin を収集している。ブログは http://blogs.gnome.org/danni/ である。

Adam Marcus (NoSQL): Adam は博士課程の学生で、データベースシステムとソーシャルコンピューティングの共通部分を MIT コンピュータ科学・人工知能研究所で研究している。最近の研究内容は、伝統的なデータベースシステムと Twitter のようなソーシャルストリーム・Mechanical Turk のようなヒューマンコンピューティング環境との関係だ。研究用のプロトタイプを便利なオープンソースシステムに仕上げるのが好き。オープンソースのストレージシステムを追いかけているほうがビーチを歩くよりも好き。ブログは http://blog.marcua.net である。

Kenneth Martin (CMake): Ken は現在 Kitware, Inc. の会長と CFO を務める。Kitware, Inc. は米国に基盤をおく研究開発会社である。彼は Kitware を 1998 年に立ち上げた共同創業者であり、会社を現在のポジションに引き上げるのに貢献した。今や同社は一流の R&D プロバイダであり、政府機関や商業関係などさまざまな分野にまたがるクライアントを抱えている。

Aaron Mavrinac (Thousand Parsec): Aaron は電子工学とコンピュータ工学をウィンザー大学で学ぶ博士候補で、カメラネットワークやコンピュータビジョン、そしてロボット工学を研究している。空き時間には、Thousand Parsec やその他のフリーソフトウェアに関する活動をしたり Python や C のコードを書いたりその他さまざまなことに手を出している。彼のウェブサイトは http://www.mavrinac.com である。

Kim Moir (Eclipse): Kim はオタワにある IBM Rational Software の研究所で Eclipse や Equinox プロジェクトのリリースエンジニアリングを率いる。また Eclipse Architecture Council のメンバーでもある。彼女が興味を持っている分野は、ビルドの最適化や Equinox そしてコンポーネントベースのソフトウェアを作ることだ。オフのときにはランニング仲間と道路を走り、次のロードレースに備えている。彼女のブログは http://relengofthenerds.blogspot.com/ である。

Dirkjan Ochtman (Mercurial): Dirkjan は 2010 年にコンピュータサイエンスの修士課程を修了した。金融関係のスタートアップ企業での勤務経験は 3 年になる。自由な時間ができると、Mercurial や Python、Gentoo Linux そして Python の CouchDB ライブラリをハックする。彼はアムステルダムの美しい都市に住んでいる。個人サイトは http://dirkjan.ochtman.nl/ である。

Sanjay Radia (HDFS): Sanjay は Yahoo!で Hadoop プロジェクトのアーキテクトを務める。Hadoop のコミッターであり、Apache Software Foundation の Project Management Committee のメンバーでもある。かつては Cassatt や Sun Microsystems そして INRIA に勤務していた経験もあり、分散システムやグリッドコンピューティング基盤の開発に携わっていた。Sanjay は、カナダのウォータールー大学でコンピュータサイエンスの博士号を取得している。

Chet Ramey (Bash): Chet は 20 年以上 bash にかかわっており、過去 17 年はメイン開発者だった。オハイオ州クリーブランドにあるケース・ウェスタン・リザーブ大学の永年勤続者である彼は、学士号と修士号もそこで取得した。クリーブランド近郊に家族やペットとともに住み、オンラインでは `http://tiswww.cwru.edu/~chet` にいる。

Emanuele Santos (VisTrails): Emanuele はユタ大学で研究をする科学者で、研究分野は科学データ管理や視覚化、起源である。彼女は 2010 年に、ユタ大学でコンピューティングの博士号を取得している。彼女は VisTrails システムのリード開発者でもある。

Carlos Scheidegger (VisTrails): Carlos はユタ大学でコンピューティングの博士号を取得し、今は AT&T Labs の研究部門で研究者として働いている。2007 年の IEEE Visualization と 2008 年の Shape Modeling International では最優秀論文に選ばれた。彼の研究分野は、データの視覚化と解析やジオメトリ処理、そしてコンピュータグラフィックスだ。

Will Schroeder (VTK): Will は Kitware, Inc. の社長で共同創業者でもある。コンピュータサイエンスの教育を受けており、VTK の主要な開発者のひとりだ。彼は美しいコードを書くことを好む。特に計算幾何学やグラフィックに関するコードでは。

Margo Seltzer (Berkeley DB): Margo はハーバード工学・応用科学大学院でコンピュータサイエンスの教授を務め、Oracle Corporation でアーキテクトとしても働いている。彼女は Berkeley DB の設計者のひとりであり、Sleepycat Software の共同創業者でもある。彼女の研究分野は、ファイルシステムやデータベースシステム、トランザクションシステム、そして医療データマイニングである。研究者としての顔は `http://www.eecs.harvard.edu/~margo` で見られ、ブログは `http://mis-misinformation.blogspot.com/` にある。

Justin Sheehy (Riak): Justin は Basho Technologies の CTO。同社は Webmachine や Riak の制作にかかわっている。Basho の前職は、MITRE Corporation の科学者そして Akamai のシステム基盤担当シニアアーキテクトだった。両社で彼が力を注いでいたのは堅牢な分散システムに関するさまざまな内容だった。スケジューリングのアルゴリズムや言語ベースの形式モデル、そして弾性などが含まれる。

Richard Shimooka (Battle for Wesnoth): Richard は、オンタリオ州キングストンにあるクイーンズ大学の Defence Management Studies Program で研究員を務めている。彼はまた、Battle for Wesnoth の管理者代理かつ長官でもある。Richard の著作には、ソーシャルグループ (政府からオープンソースプロジェクトまでの幅広いもの) の組織文化に関して調査したものがいくつかある。

Konstantin V. Shvachko (HDFS) はベテランの HDFS 開発者で、eBay の Hadoop アーキテクトのリーダーである。Konstantin の専門分野は、大規模な分散ストレージシステムのための効率的なデータ構造やアルゴリズムである。彼は平衡木の新しい方式である S-tree を考案した。これは構造化されていないデータの索引付けに最適化されたものだ。また彼は、S-tree ベースの Linux ファイルシステムである treeFS の初期の開発者だった。treeFS は、後の reiserFS の原型となった。Konstantin は、ロシアのモスクワ大学でコンピュータサイエンスの博士号を取得している。また、Apache Hadoop の Project Management Committee のメンバーでもある。

Claudio Silva (VisTrails): Claudio は、ユタ大学のコンピュータサイエンスの正教授である。

彼の研究分野は、視覚化や幾何学的コンピューティング、コンピュータグラフィックス、そして科学データ管理などだ。彼は 1996 年に、ニューヨーク州立大学ストーニブルック校でコンピュータサイエンスの博士号を取得した。2011 年後半には、ニューヨーク大学ポリテクニック研究室にコンピュータサイエンスおよびエンジニアリングの正教授として合流する予定だ。

Suresh Srinivas (HDFS): Suresh は、Yahoo!のソフトウェアアーキテクトとして HDFS にかかわっている。彼は Hadoop のコミッターであり、Apache Software Foundation の PMC のメンバーでもある。Yahoo!の前には Sylantro Systems で働いており、コミュニケーションサービスのホスティングのためのスケーラブルな基盤を開発していた。Suresh は、インドのカルナタカにあるナショナル工科大学でエレクトロニクスと通信の学位を取得した。

Simon Stewart (Selenium): Simon はロンドン在住で、Google のソフトウェアテストエンジニアとして働いている。彼は Selenium プロジェクトの主要な貢献者である。WebDriver の作者でもある彼は、オープンソースにほれ込んでいる。Simon が好きなのはビールを楽しむこととソフトウェアを書くことで、ときにはそれらを同時にすることもある。個人ホームページは `http://www.pubbitch.org/` である。

Audrey Tang (SocialCalc): Audrey は、台湾在住のプログラマーであり翻訳家でもある。現在の勤務先は Socialtext で、彼女のそこでの役職は “Untitled Page” だ。また、Apple のローカライズやリリースエンジニアリングも請け負っている。彼女はかつて Pugs プロジェクトを率いていた。これは実際に動作する Perl 6 の初めての実装だった。また、CPAN や Hackage にも多大な貢献をしている。彼女のブログは `http://pugs.blogs.com/audreyt/` である。

Huy T. Vo (VisTrails): Huy は、2011 年 5 月にユタ大学で博士号を取得した。彼の研究分野は、視覚化やデータフローアーキテクチャ、そして科学データ管理などである。VisTrails, Inc. で上級開発者として働く。彼はまた、ニューヨーク大学ポリテクニック研究室で博士研究員となることも決まっている。

David White (Battle for Wesnoth): David は、Battle for Wesnoth の創設者でありリード開発者である。David はこれまでにもいくつかのオープンソースビデオゲームプロジェクトにかかわってきた。共同で立ち上げた Frogatto もそのひとつだ。彼は Sabre Holdings のパフォーマンスエンジニアであり、旅行技術のリーダーでもある。

*Greg Wilson (**編集担当**)*: Greg は過去 25 年にわたって高性能科学計算やデータの視覚化、コンピュータセキュリティなどにかかわってきた。数冊のコンピュータ関連書籍 (2008 年の Jolt Award を受賞した *Beautiful Code* など) に著者あるいは編集者としてかかわっており、こども向けの本も二冊出版している。Greg は 1993 年にエジンバラ大学でコンピュータサイエンスの博士号を取得した。

Tarek Ziadé (Python Packaging): Tarek はフランスのブルゴーニュに住む。Mozilla の上級ソフトウェアエンジニアであり、サーバーを Python で構築している。空き時間には、Python のパッケージングを率いている。

謝辞

レビュアーのみなさんに感謝する。

Eric Aderhold	Muhammad Ali	Lillian Angel
Robert Beghian	Taavi Burns	Luis Pedro Coelho
David Cooper	Mauricio de Simone	Jonathan Deber
Patrick Dubroy	Igor Foox	Alecia Fowler
Marcus Hanwell	Johan Harjono	Vivek Lakshmanan
Greg Lapouchnian	Laurie MacDougall Sookraj	Josh McCarthy
Jason Montojo	Colin Morris	Christian Muise
Victor Ng	Nikita Pchelin	Andrew Petersen
Andrey Petrov	Tom Plaskon	Pascal Rapicault
Todd Ritchie	Samar Sabie	Misa Sakamoto
David Scannell	Clara Severino	Tim Smith
Kyle Spaans	Sana Tapal	Tony Targonski
Miles Thibault	David Wright	Tina Yee

また、編集の初期段階での助けとなった Jackie Carter にも感謝する。

貢献

何十人ものボランティアのおかげで本書を作ることができたが、まだやり残したことは多い。間違いの指摘、他の言語への翻訳、他のオープンソースプロジェクトのアーキテクチャに関する記述の追加などを歓迎する。協力してくれる場合は `aosa@aosabook.org` まで連絡してほしい。

第1章

Asterisk

Russell Bryant

Asterisk[1] は GPLv2 で配布されているオープンソースによる電話通信のプラットフォームである。簡単に言うと、電話を掛けたり、受けたり、カスタム処理を実行したりするためのサーバアプリケーションである。

このプロジェクトは 1999 年に Mark Spencer によって始められた。Mark は自らが興した Linux Support Services 社において、電話システムが必要だったのだが、購入するだけの費用が無かったために自作した。Asterisk の人気が上がると、Asterisk に資源を集中するため社名を Digium, Inc に改めた。

Asterisk の名前の由来は、UNIX におけるワイルドカード文字*にある。Asterisk プロジェクトの目標はすべての電話テクノロジを実装する事である。この目標に向けて、Asterisk は、電話を掛けたり、受けたりするための多くのテクノロジをサポートしている。これらのテクノロジには、従来のアナログおよびデジタルの電話ネットワークである PSTN(Public Switched Telephone Network) はもちろん、多くの VoIP(Voice over IP) プロトコルも含まれる。このような異なる種類の電話テクノロジを実装していること、異なる電話テクノロジ同士を接続できる事が Asterisk の主な強みである。

Asterisk システムに電話が掛かってきたり、Asterisk から電話を掛けたりするとき、通話処理をカスタマイズするのに使える多くの追加機能がある。ボイスメールのような完成されたアプリケーションもある一方、音声ファイルを流したり、数字を読んだり、音声認識といった、組み合せて使うことでカスタムの音声アプリケーションを構築するのに使えるようなより小規模の機能群もある。

1.1 重要なアーキテクチャ・コンセプト

このセクションでは、Asterisk 全体に影響するアーキテクチャのコンセプトについて述べる。ここで述べる考え方は、Asterisk アーキテクチャの根幹を構成する。

[1]http://www.asterisk.org/

チャネル

Asterisk におけるチャネルは、Asterisk システムと電話端末の接続をあらわす (図 1.1)。最も単純な例は、電話機が Asterisk システムを呼び出す時である。この場合の接続は、一つのチャネルであらわされる。Asterisk のコードでは、チャネルは ast_channel 構造体のインスタンスとして存在する。この呼び出しシナリオは、発信者がボイスメールを使うときの例である。

図 1.1: 片方向通話, 単一チャネル

チャネル・ブリッジ

おそらく、より馴染のある通話シナリオは電話機同士の接続であろう。このシナリオでは、電話機 A を使っている人が電話機 B を呼び出す。すなわち 2 台の電話端末が Asterisk システムに接続しているため、二つのチャネルが存在する (図 1.2)。

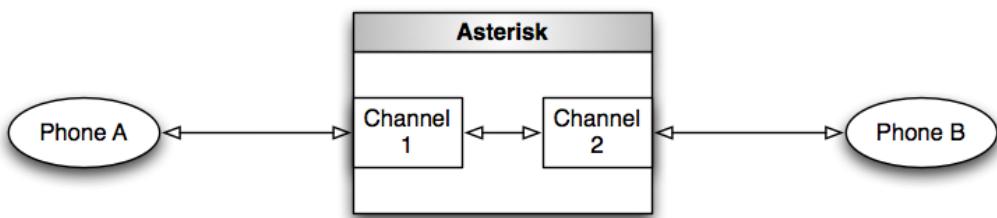



図 1.2: 2 つのチャネルによって表される双方向通話

このように Asterisk チャネルが接続される事をチャネル・ブリッジと呼ぶ。チャネル・ブリッジは、チャネル同士をつなげて、メディアを流す事を目的としている。流れるメディアは音声が主になるが、ビデオやテキストも可能であるし、ひとつ以上のメディア（音声とビデオなど）のこともある。このような場合も、Asterisk では、一つのチャネルとして扱われ

る。図 1.2 では、２つのチャネルがそれぞれ、電話 A、B に接続されている。このブリッジは、電話機 A から電話機 B へのメディア・ストリームおよび電話機 B から電話機 A へのメディア・ストリームを通すことに責任を持つ。すべてのメディア・ストリームは、Asterisk と交渉されるため、Asterisk が理解できない、またはフルにコントロールできないメディア・ストリームは許可されない。これは、Asterisk において録音や音声処理、異なるテクノロジ間の変換が可能であることを意味する。

２つのチャネルをブリッジにより接続する形式は２通りある。ジェネリック・ブリッジとネイティブ・ブリッジである。ジェネリック・ブリッジは、チャネル・テクノロジに何が使われていようが動作する。すべての音声とシグナリングは、Asterisk 抽象チャネル・インタフェースを通してやりとりされる。これは、もっとも柔軟性の高い接続であるが、実現するために高いレベルの抽象化が必要であり、効率性を犠牲にしている。図 1.2 は、ジェネリック・ブリッジを表現している。

ネイティブ・ブリッジは、テクノロジ固有のチャネル同士を接続するときに使われる手法である。２つのチャネルが、同じメディア・トランスポート・テクノロジを使って Asterisk に接続されるときには、異なるテクノロジ同士を接続する時に使用する Asterisk 抽象レイヤを通しての接続よりも、より効果的な接続方法がある。例えば、電話ネットワークに接続するのに特別のハードウェアが使われるときは、アプリケーションを全く通らずにチャネル同士をハードウェア上でブリッジする事が可能である。いくつかの VoIP プロトコルの場合には、呼制御信号の情報はサーバを通して流れるが、メディア・ストリームはエンドポイント間で相互に直接送受信させることも可能である。

ジェネリック・ブリッジかネイティブ・ブリッジかの選択は、ブリッジが必要になったときに、２つのチャネルの比較で決定される。二つのチャネルが同じネイティブ・ブリッジ技術をサポートする事がわかると、ネイティブ・ブリッジが使われる。その他の場合は、ジェネリック・ブリッジが使われる。２つのチャネルが同じネイティブ・ブリッジをサポートするかどうかの決定は、単純に C 言語の関数ポインタの比較で行われる。これは、もっともエレガントな手法というわけではないが、この方法が我々のニーズを満たさなかったことは今のところない。ネイティブ・ブリッジ機能については、1.2 節で詳しく説明する。図 1.3 は、ネイティブ・ブリッジの例を表現している。

フレーム

通話中のコミュニケーションは、Asterisk のコードではフレームで表現されている。これは、`ast_frame` 構造体のインスタンスである。フレームはメディア・フレームにもシグナリング・フレームにも使われる。音声メディア・フレームのストリームは基本的にシステムを通過する。シグナリング・フレームは、押された番号や保留中、呼切断などの呼制御イベントを送信するのに使われる。

利用可能なフレーム・タイプのリストは静的に定義される。フレーム・タイプは、タイプおよびサブタイプで記されて、数値の形式で保存される。フレーム・タイプの全リストはソー

図 1.3: ネイティブ・ブリッジの例

スコードの include/asterisk/frame.h にある。代表的な例を以下に示す。

- VOICE: オーディオ・ストリーム・フレーム
- VIDEO: ビデオ・ストリーム・フレーム
- MODEM: IP 上で FAX を送信するための T.38 のようなデータに対する符号化を表わす。このフレーム・タイプの主な使用対象は FAX の処理である。この信号が相手側で正しく復号できるように、フレームのデータが絶対に変更されないことが重要である。これは、AUDIO フレームが帯域幅を稼ぐために音声品質を犠牲にしてコーデックを変換することが許されるという点で異なる。
- CONTROL: 呼制御メッセージを表わす。このフレームは、呼制御イベントを示すのに使われる。これらのイベントには、応答や切断、保留などがある。
- DTMF_BEGIN: 番号の始まり。このフレームは、発信者が DTMF キー [2] を押したとき送信される。
- DTMF_END: 番号の終わり。このフレームは、発信者が DTMF キーを押し終わったときに送信される。

1.2 Asterisk 抽象コンポーネント

Asterisk は高度にモジュール化が進んだアプリケーションである。ソースツリー中の main/ディレクトリ以下にコア・アプリケーションがある。しかし、これ自体は、大変有益というわけではない。コア・アプリケーションは、基本的にモジュール・レジストリとして振る舞う。通話をうまく動かすための抽象インターフェースとの接続用のコードも存在する。これらインタフェースの具体的な実装は実行時にローダブルモジュールによって登録される。

[2] DTMF は Dual-Tone Multi-Frequency を表わす。これは電話機のキーを押したときにオーディオの形式で送信されるトーン信号である。

デフォルトでは、メイン・アプリケーションが起動するときに、ファイルシステム上のあらかじめ定められた場所にある全ての Asterisk モジュールがロードされる。このアプローチは、簡素化のために導入された。しかし、はっきりとロードするモジュールをどの順番でロードするかを指定することができるコンフィグレーション・ファイルもある。これは、コンフィグレーションを少し複雑にするが、不要なモジュールをロードしないように指定できる能力を提供する。このことは、アプリケーションのメモリ・フットプリント削減が主な利点であるが、セキュリティ面での利点もいくつかある。ネットワーク接続に必要なモジュール以外は、ロードしないのがベストである。

モジュールはロードされるときに Asterisk コア・アプリケーションに抽象コンポーネントの実装を登録する。モジュールが Asterisk コアに対して実装および登録できるインタフェースはたくさんある。モジュールは自身が登録したいインタフェースは種別が異なっていれば全て登録することができる。一般的に、関連する機能は一つのモジュールにまとめられる。

チャネル・ドライバ

Asterisk チャネル・ドライバのインタフェースは、最も複雑かつ重要である。Asterisk チャネル API は、電話プロトコルの抽象化を提供する。抽象化により使用する電話プロトコルと独立して Asterisk 機能が動けるようになる。このコンポーネントは、Asterisk 抽象チャネルと実行される電話テクノロジの細部の間を通訳する責務を持つ。

Asterisk チャネル・ドライバ・インタフェースの定義は `ast_channel_tech` インタフェースと呼ばれる。これは、チャネル・ドライバによって実装されなければならない一連のメソッドを定義する。チャネル・ドライバが実装しなければならない最初のメソッドは、`ast_channel` ファクトリ・メソッドであり、具体的には `ast_channel_tech` の中の `requester` メソッドである。Asterisk チャネルが生成されるとき、これが受話であろうが発話であろうが、要求されたチャネルの種類に対応した `ast_channel_tech` の実装は、この電話に対する `ast_channel` をインスタンス化、初期化する責務を持つ。

`ast_channel` が生成される時、生成元の `ast_channel_tech` への参照も作られる。テクノロジ固有の方法で扱われるべき多くのオペレーションがある。これらのオペレーションが `ast_channel` のなかで実行されるときには、オペレーションのハンドリングは、`ast_channel_tech` の適切なメソッドに委ねられる。 図 1.2 は、2 つの Asterisk チャネルを示している。 図 1.4 は、これを展開して、二つのブリッジされたチャネルとチャネル・テクノロジの実装がどのようにかみ合うのかを図示している。

`ast_channel_tech` の中で最も重要なメソッドを以下に示す。

- `requester`: このコールバックは、チャネル・ドライバにチャネルタイプに対して適切な `ast_channel` オブジェクトのインスタンス化と初期化を要求する。
- `call`: このコールバックは、`ast_channel` に示されているエンドポイントへの発信を開始するのに使われる。

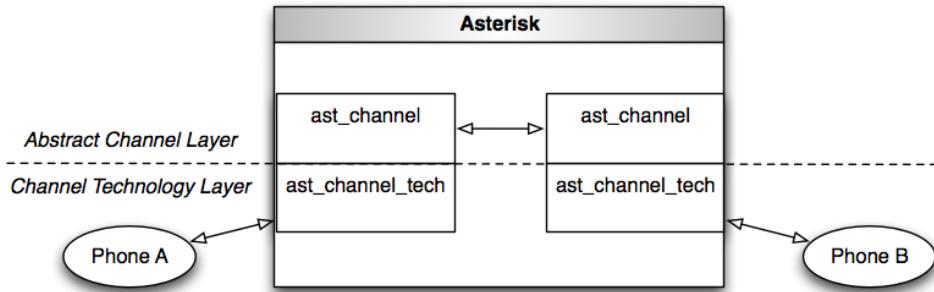



図 1.4: チャネル・テクノロジと抽象チャネル・レイヤ

- answer: Asterisk が本 ast_channel への着信に応答すると決めたときに呼ばれる。
- hangup: システムが呼切断すると決めたときに呼ばれる。チャネル・ドライバは、通話が終わったことをプロトコル特有の方法でエンドポイントに伝える。
- indicate: 通話が始まると、エンドポイントに伝えるべき多くの制御イベントが発生する。例えば、デバイスが保留になると、この状態を伝えるために本コールバックが呼ばれる。通話が保留になった事を示す方法は、プロトコル毎に異なる。チャネル・ドライバは、デバイスに保留音を流し始めるだけの事もある。
- send_digit_begin: この関数はデバイスへの数字 (DTMF) 送信が始まった事を示すときに呼ばれる。
- send_digit_end: この関数はデバイスへの数字 (DTMF) 送信が終わった事を示すときに呼ばれる。
- read: この関数は、デバイスが送った ast_frame をリードするために Asterisk コアによって呼ばれる。ast_frame は、メディア (オーディオ、ビデオなど) や呼制御信号をカプセル化するのに Asterisk で使用される抽象化である。
- write: この関数は、デバイスに ast_frame を送信するのに使われる。チャネル・ドライバは、データを取得して電話プロトコルに対応した形に、パケット化してエンドポイントに送信する。
- bridge: このチャネル・タイプに対するネイティブ・ブリッジのコールバックである。ネイティブ・ブリッジは、前に述べたように、2つのチャネルが同じ種類のときに、呼制御信号やメディアが不要な抽象レイヤを流れるかわりに、チャネル・ドライバがより効率的なブリッジ方法を実行できるときに使用する。これは、性能面で大変重要である。

通話が終了すると、Asterisk コア内の抽象チャネルを扱うコードは、ast_channel_tech hangup コールバックを起動し ast_channel オブジェクトを破棄する。

ダイヤルプラン・アプリケーション

Asterisk アドミニストレータは、Asterisk ダイヤルプランを使って通話手順を設定する。ダイヤルプランは/etc/asterisk/extensions.conf に存在する。ダイヤルプランはエクステンションと呼ばれる一連の通話ルールを構成する。呼び出しがシステムに到着すると、呼び出し処理をするために、ダイヤル番号を使ってダイヤルプラン内の対応するエクステンションを探索する。エクステンションは、チャネル上で実行するダイヤルプラン・アプリケーションのリストを持っている。ダイヤルプランで実行可能なアプリケーションは、アプリケーション・レジストリが保持している。このレジストリはモジュールがロードされるときに登録される。

Asterisk は 200 近いアプリケーションを持つ。アプリケーションの定義は大変ルーズである。アプリケーションはチャネルとインタラクトするために Asterisk の内部 API を自由に使うことができる。発信者にサウンド・ファイルを流す Playback のような単純なタスクを行うアプリケーションもあるし、 Voicemail のようなより複雑で大規模なアプリケーションもある。

Asterisk ダイヤルプランを使って複数のアプリケーションを組み合せると、カスタムの通話処理が可能である。提供されたダイヤルプラン言語で実現できるカスタム化よりも複雑なものが必要なときには、都合の良いプログラミング言語を使ってカスタムの通話処理が可能なスクリプト・インタフェースもある。これらのスクリプトのためインタフェースを使って他のプログラム言語が使われる時でも、ダイヤルプラン・アプリケーションはチャネルと相互作用するために起動される。

次の例に入る前に、番号 1234 への通話を扱う Asterisk ダイヤルプランの文法について説明する。 1234 は無作為の選択である。これは、３つのダイヤルプラン・アプリケーションを起動する。最初に、呼応答し、次に、サウンド・ファイルを再生し、最後に呼切断を行う。

```
; Define the rules for what happens when someone dials 1234.
;
exten => 1234,1,Answer()
    same => n,Playback(demo-congrats)
    same => n,Hangup()
```

exten キーワードはエクステンションを定義するのに使われる。exten 行の右側において、1234 は誰かが 1234 を呼び出したときのルールを定義している。次の 1 は、この番号にダイヤルされたときに最初に実行されるステップである事を示している。最後に、Answer はシステムが呼に応答することを指示する。次の same キーワードで始まる 2 行は、直前に指定したエクステンションと同じであることを示す。本例では、1234 に対するエクステンションである事を示している。n は、次のステップであることを示す短縮表現である。各行の最後の項目には実行する動作を指定する。

次は、Asterisk ダイヤルプランを使った別の例である。このケースは次のような流れである。着呼に対して応答する。発信者にビープ音を鳴らし、発信者からの数字を 4 桁読み、DIGITS

変数に格納される。さらに、格納された 4 桁の数字が、発信者に音声で通知される。最後に終話する。

```
exten => 5678,1,Answer()
    same => n,Read(DIGITS,beep,4)
    same => n,SayDigits(${DIGITS})
    same => n,Hangup()
```

前に触れたように、アプリケーションの定義は、大変ルーズである。登録された関数プロトタイプは非常に単純である：

```
int (*execute)(struct ast_channel *chan, const char *args);
```

しかしながら、アプリケーションの実装は、実際上、`include/asterisk/`にある全ての APIs を使用する。

ダイヤルプラン・ファンクション

多くのダイヤルプラン・アプリケーションは、文字列型の引数をとる。決め打ちの場合もあるし、動的な振る舞いの場合には、変数も使われる。次の例は、変数を設定して、`Verbose` アプリケーションを使って、その値を Asterisk コマンドライン・インタフェースに表示するダイヤルプランの断片である。

```
exten => 1234,1,Set(MY_VARIABLE=foo)
    same => n,Verbose(MY_VARIABLE is ${MY_VARIABLE})
```

ダイヤルプラン・ファンクションは、前の例と同じ構文を使って起動される。Astersisk モジュールは、ダイヤルプラン・ファンクションを登録することができる。ダイヤルプラン・ファンクションは、情報を引き出してダイヤルプランに反映させることができる。ダイヤルプラン・ファンクションは、ダイヤルプランからデータを受けて実行する事もできる。一般的なルールとして、ダイヤルプラン・ファンクションは、チャネル・メタデータをセットしたり引き出したりはできるが、呼制御やメディア処理は行わない。これは、ダイヤルプラン・アプリケーションの仕事として残されている。

次は、ダイヤルプラン・ファンクションの利用例である。最初に、現在のチャネルの発信者 ID を Asterisk コマンドライン・インタフェースに表示する。次に `Set` アプリケーションを使って発信者 ID を変更する。この例では、`Verbose` と `Set` はアプリケーションであり、`CALLERID` がファンクションである。

```
exten => 1234,1,Verbose(The current CallerID is ${CALLERID(num)})
    same => n,Set(CALLERID(num)=<256>555-1212)
```

ここでは、発信者番号情報が、`ast_channel` のインスタンスのデータ構造体に保存されているため、ダイヤルプラン・ファンクションが必要であった。ダイヤルプラン・ファンクショ

ン・コードは、これらのデータ構造体にデータを設定したり引き出したりする方法を知っている。

　もうひとつのダイヤルプラン・ファンクションの例は、通話記録にカスタム情報を加える。これは、CDR(Call Detail Records) と呼ばれる。CDR ファンクションは、通話詳細記録情報の引き出しやカスタム情報の追加を可能にする。

```
exten => 555,1,Verbose(Time this call started: ${CDR(start)})
    same => n,Set(CDR(mycustomfield)=snickerdoodle)
```

符号化方式変換

VOIP の世界では、異なる符号媒体のネットワークをまたがって送信するために、様々な種類の符号化方式が使われる。符号化方式は、メディアの品質、CPU 消費量、帯域幅のトレードオフの中から選択される。Asterisk は多くの符号化方式をサポートし、必要ならば、これらの符号化方式を相互に変換可能である。

通話がセットアップされると、Asterisk は符号化方式変換が不要になるように両端の端末に共通の符号化方式を選択することを試みる。しかし、必ずしも可能というわけではない。共通の符号化方式が使われていても、符号化方式変換が使われる事もある。例えば、Asterisk は、音声がシステムを通過するときに (ボリューム調節などの) 信号処理を行うように構成することも可能である。このときに、Asterisk は、信号処理を行う前に、音声を非圧縮形式に変換する必要がある。Asterisk は、通話録音も可能である。録音に指定した符号化方式が通話の符号化方式と異なるときは、符号化方式変換が必要となる。

符号化方式交渉

メディア・ストリームにどの符号化方式を使用するかの交渉は、Asterisk に通話を接続するテクノロジに依存する。従来の電話通信ネットワーク (いわゆる PSTN）上の通話のようなケースでは、交渉の余地は無いと思われる。しかし、特に IP プロトコルを使うようなケースでは、利用可能な符号化方式や優先度を示して、符号化方式を交渉する機構により、符号化方式の合意が取られる。
例えば、SIP(最も一般的な VOIP プロトコル) の場合、通話が Asterisk に届いたときに、符号化方式の交渉が行われる。

1. 端末が Asterisk に通話要求を送信するときに、使いたい符号化方式のリストも含める。
2. Asterisk は、アドミニストレータが用意した、優先度順に並んだ利用可能な符号化方式のリストを参照する。Asterisk はこのリストと端末の要求したリストから最も好ましい符号化方式を選択して応答する。

Asterisk が充分に扱えない分野のひとつに、ビデオのようなより複雑な符号化方式がある。過去 10 年で、符号化方式の交渉に対する要求は、より複雑化してきた。最新の音声符号化方式や、サポートするビデオ符号化方式の改善には、多くの作業が残っている。これは、Asterisk の次のメジャーリリースに向けた開発作業の最優先項目のひとつである。

符号化方式変換モジュールは、一つ以上の `ast_translator` インタフェースの実装を利用する。変換モジュールは、変換元と変換先の属性を持つ。また、変換元フォーマットから変換先フォーマットへのメディア・チャンクの変換に使われるコールバックも実装する。変換

モジュールは、電話の概念については、関知せず、メディアの変換のみに関わる。

符号化方式変換APIのより詳細情報は、include/asterisk/translate.hとmain/translate.cにある。符号化方式変換の抽象化部の実装は、codecs ディレクトリにある。

1.3 スレッド

Asterisk は、マルチスレッドを多用するアプリケーションである。Asterisk は、スレッドを管理するのに POSIX スレッド API とロックなどの関連サービスを使っている。スレッドを扱うすべての Asterisk のコードは、デバッグ目的で、一連のラッパーを通している。Asterisk 内のほとんどのスレッドは、ネットワーク監視スレッドか、チャネルスレッド (PBX スレッドとも呼ばれる。主目的がチャネルに対する PBX 機能の実行である事に起因する）に分類される。

ネットワーク監視スレッド

ネットワーク監視スレッドは、Asterisk の主要チャネル・ドライバ毎に存在する。ネットワーク監視スレッドは、接続されたネットワーク (IP や PSTN など) を監視し、呼着信や他の着信要求を監視する。本スレッドは、コネクションの初期セットアップを行い、認証やダイヤルされた番号の検証を行う。呼のセットアップが完了すると、監視スレッドは、Asterisk チャネル (ast_channel) のインスタンスを生成し、チャネル・スレッドを始動させて呼の切断までの残りの制御を行わせる。

チャネル・スレッド

前に述べたように、チャネルは、Asterisk の基本概念である。チャネルは、着信にも発信にも対応する。着信チャネルは、呼が Asterisk システムに届いたときに生成される。これらのチャネルは、Asterisk ダイヤルプランを実行する。ダイヤルプランを実行する着信チャネル毎に、スレッドが生成される。これらのスレッドは、チャネル・スレッドと呼ばれる。

ダイヤルプラン・アプリケーションは、常にチャネル・スレッドのコンテキストで実行される。ダイヤルプラン・ファンクションも**ほとんど**常に、チャネル・スレッドのコンテキストで実行される。Asterisk CLI のような非同期インタフェースからダイヤルプラン・ファンクションを読んだり書いたりする事が可能である。しかし、ast_channel データ構造の所有者は、常にチャネル・スレッドであり、同スレッドが ast_channel オブジェクトの生成・消滅もコントロールする。

1.4　通話シナリオ

前の 2 節では、Asterisk コンポーネントに対する重要なインタフェースとスレッド実行モデルを紹介した。本節では、複数の Asterisk コンポーネントがどのように協調して通話処理するのかを示すために、いくつかの一般的な通話シナリオに分けて示す。

ボイスメールのチェック

一例として、誰かが電話システムを呼び出して、ボイスメールをチェックする通話シナリオを示す。本シナリオでの最初の主要コンポーネントは、チャネル・ドライバである。チャネル・ドライバは、電話からの着呼要求の処理に責任を持つ。これは、チャネル・ドライバの監視スレッドで行われる。呼をシステムに運ぶのに使われる電話テクノロジによっては、通話をセットアップするのに必要な交渉が行われる事もある。呼のセットアップのもう一つのステップは、通話先の決定である。これは、通常、発信者がダイヤルした番号により決定される。しかし、あるケースでは、通話の伝送テクノロジがダイヤル番号の明示をサポートしないため、番号特定ができない事もある。アナログ電話の着信が一つの例である。

ダイヤルプラン中のエクステンションにダイヤルされた番号が存在することをチャネル・ドライバが確認すると、チャネル・ドライバは Asterisk チャネル・オブジェクト (`ast_channel`) を割り当て、チャネル・スレッドを起動する。チャネル・スレッドは、残りの通話処理の責任を任せられる (図 1.5)。

チャネル・スレッドのメインループは、ダイヤルプランの実行を扱う。ダイヤルされたエクステンションに対して定義されたルールを探索して、定義されたステップに従って順次実行する。次に示すエクステンションの例は、`extensions.conf` ダイヤルプラン内で。このエクステンションは、誰かが、`*123` にダイヤルしたときに、呼び出しに応答し、`VoicemailMain` アプリケーションを実行する。このアプリケーションは、ユーザが自分宛のメールボックスに残されたメッセージをチェックするものである。

```
exten => *123,1,Answer()
    same => n,VoicemailMain()
```

チャネル・スレッドが `Answer` アプリケーションを実行すると、Asterisk は着信に応答する。呼び出しに応答するには、テクノロジ固有の処理を必要とする。よって、いくつかの一般的な応答処理に加えて、`ast_channel_tech` 構造体に関連した `answer` コールバックが応答処理のために呼ばれる。これは、IP ネットワーク上に特定のパケットを送信する処理やアナログ電話でのオフフック処理などである。

次のステップでは、チャネル・スレッドは、`VoicemailMain`(図 1.6) を実行する。このアプリケーションは、`app_voicemail` モジュールによって供給される。特筆するべき点は、ボイスメールのコードが通話に応対している間、ボイスメール自体は、Asterisk システムへの呼

図 1.5: 呼設定シーケンス図

び出しを伝送するテクノロジについては何も知らないということである。Asterisk 抽象チャネルは、ボイスメールの実行処理から、これらの詳細を隠蔽している。

発信者が、自身のボイスメールにアクセスする処理には多くの機能が含まれる。しかし、基本的には、発信者からの数字キーの入力に応答して、サウンド・ファイルを読み込んだり、書き込んだりするような処理である。DTMF 信号は、多くの異なる方式で Asterisk に運ばれる。繰り返しになるが、これらの詳細方式の部分は、チャネル・ドライバで行われる。キー入力が Asterisk に届くと、共通のキー入力イベントに変換されてからボイスメールのコードに届く。

これまで述べてきた中で、Asterisk の主要インターフェースの一つは、符号化方式変換である。これらの符号化の実装は、この呼び出しシナリオでは大変重要である。ボイスメールのコードが発信者に対してサウンド・ファイルを再生したいとき、サウンド・ファイルのオー

ディオ形式は、Asterisk と発信者の間で使われているフォーマットと異なるときもある。オーディオ形式を変換する必要がある時、一つ以上の変換器を使って、変換元から変換先の形式への符号化変換のパスを構築する。

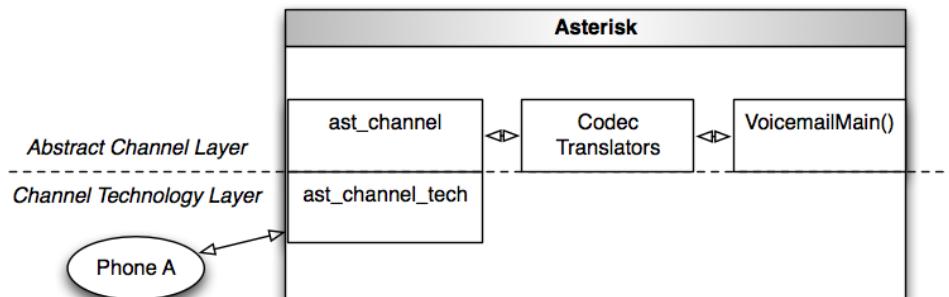



図 1.6: `VoicemailMain` への呼び出し

どこかの時点で発信者は、ボイスメール・システムとやり取りして呼切断を行う。チャネル・ドライバは、これを検出して Asterisk チャネル呼制御の共通イベントへと変換する。この呼制御イベントを受信すると、ボイスメールのコードは残り作業が無いため終了する。制御がチャネル・スレッドのメインループに戻って、ダイヤルプランの実行を続ける。この例では、これ以上のダイヤルプラン処理がないため、チャネル・ドライバはテクノロジ固有の呼切断処理を与えられる。これにより、`ast_channel` オブジェクトは破棄される。

ブリッジ・コール

Asterisk における、もう一つの、よくある通話シナリオは、二つのチャネル間のブリッジ・コールである。これは 2 者間での通話のシナリオである。呼設定処理の最初の部分は、前例と同じである。処理の違いは、通話が設定されて、チャネル・スレッドが、ダイヤルプランを実行し始めるときから生じる。

次のダイヤルプランは、ブリッジ・コールになる典型的な例である。このエクステンションを使うと、電話機が、`1234` をダイヤルすると、ダイヤルプランは、`Dial` アプリケーションを実行する。これは、発信を開始するときのメイン・アプリケーションである。

```
exten => 1234,1,Dial(SIP/bob)
```

`Dial` アプリケーションへの引数は、システムに、`SIP/bob` への発信を促す。この引数の `SIP` の部分は、通話に使われるプロトコルが SIP である事を示している。bob の部分は、SIP プロトコルを実装したチャネル・ドライバ `chan_sip` によって、解釈される。チャネル・ドライバが、bob へのアカウントを正しく設定していると仮定すると、Bob の電話機への呼の伝送方法を知ることになる。

Dial アプリケーションは、SIP/bob 識別子を使って Asterisk コアに対して新しい Asterisk チャネルを割り当てる。コアは、SIP チャネル・ドライバに対してテクノロジ固有の初期化処理を行うように指示する。チャネル・ドライバは、電話の呼び出しを行う処理を起動する。リクエストが進ときに、Asterisk コアに対してイベントの発生を知らせる。これは、さらに、Dial アプリケーションまで届けられる。これらのイベントは、呼応答、ビジー、輻輳、何らかの理由での拒絶など多くの応答種別が含まれる。理想的なケースでは、呼び出しは応答される。インバウンド・チャネルを通じて呼が応答された事は伝わる。システムが、アウトバウンドコールへの応答が完了するまで、Asterisk は、この呼に対しては応答しない。両チャネルが、応答するとチャネル・ブリッジが始まる (図 1.7)。

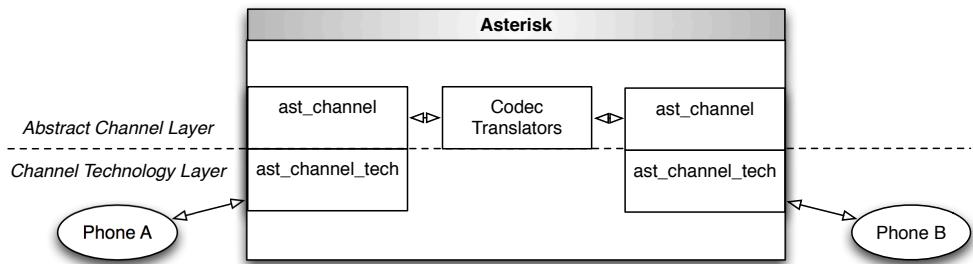



図 1.7: ジェネリック・ブリッジのブリッジ・コールにおけるブロック図

チャネル・ブリッジを行っている間、片方のチャネルからのオーディオとシグナリング・イベントは、もう一方のチャネルに通される。これは、片側からの呼切断などのブリッジの終了を示すイベントが起こるまで続けられる。図 1.8 に示したシーケンス図は、ブリッジ・コールの間オーディオ・フレームに対して実行される主な流れを表している。

通話が終わると、切断処理は前の例とほとんど同じように進められる。主な違いは、チャネルが二つあることである。チャネル・テクノロジ固有の切断処理は、チャネル・スレッドが実行を止める前に実行される。

1.5 Final Comments

Asterisk のアーキテクチャは、誕生してから 10 年を越える歳月が経っている。しかし、拡大を続ける産業においても、チャネルの基本概念と Asterisk ダイヤルプランを使った呼制御の柔軟性は、複雑な電話通信システムの開発をサポートし続けている。Aseterisk のアーキテクチャで充分でない分野のひとつに、複数のサーバに渡るスケーリングがある。Asterisk 開発コミュニティは、このスケーラビリティを改善するための Asterisk SCF(Scalable Communications Framework) と呼んでいる兄弟プロジェクトを進めている。数年内には、Asterisk と Asterisk SCF が統合されて、より大規模システムへの採用が進み、電話市場における重要性を増すと考えている。

図 1.8: ブリッジ中のオーディオ・フレーム処理のシーケンス図

第2章

Audacity

James Crook

Audacity は、よく知られたサウンドレコーダー/オーディオエディタだ。多機能なプログラムでありながら、使いやすさも維持している。大半のユーザーは Windows 上で使っているが、Audacity のソースコードをコンパイルして Linux や Mac でも使える。

Dominic Mazzoni が最初に Audacity を書いたのは 1999 年のこと。当時の彼はカーネギーメロン大学の研究生で、音響処理アルゴリズムの開発やデバッグに使うプラットフォームを作ろうとしていた。その後このソフトは成長し、当初の目的以外にもいろいろな方面で役立つようになった。Audacity がオープンソースソフトウェアとして公開されると、多くの開発者がそれにひきつけられた。熱心なファンたちが参加し、Audacity の改良や保守、テスト、更新、ドキュメント作成、ユーザーのサポートなどを行うようになった。また、長い年月をかけて、ユーザーインターフェイスも他の言語に翻訳された。

Audacity のひとつの目標は、ユーザーインターフェイスを "発見可能" なものにするということだ。特にマニュアルを読まずともすぐに使い始めることができ、使っていくうちにいろいろな機能を発見できるようになることを目指している。この方針もあり、Audacity は他のソフトに比べてユーザーインターフェイスの一貫性をより重視している。多くの人がかかわるプロジェクトでは、このような統一指針が思いのほか重要となる。

Audacity のアーキテクチャにも同様の指針や発見可能性があればどんなによいことか。それに近いこととして我々が言えるのは "試して、そして合わせよ" ということだ。新しいコードを追加するとき、開発者はその周辺のコードを見てその形式や規約に合わせようとする。しかし実際のところ、Audacity のコードベースには、しっかり構成されたコードとそうでないコードが入り混じっている。全体のアーキテクチャを、小さな都市になぞらえて考えるとわかりやすい。印象的な建造物がいくつかある一方で、そのそばには荒廃した貧民街も見受けられるという具合だ。

2.1 Audacityの構造

Audacityは、いくつかのライブラリによる階層構造になっている。Audacityのコードに新しいプログラムを追加するときにはこれらのライブラリに関する詳細な知識は不要だが、ライブラリのAPIやその役割に親しんでおくことは大切だ。中でも最も重要なライブラリはPortAudioとwxWidgetsだ。PortAudioはローレベルのオーディオインターフェイスをクロスプラットフォームで提供し、wxWidgetsはGUIコンポーネントを同じくクロスプラットフォームで提供する。

Audacityのコードを読むときには、本質的に不可欠なコードは一部だけであることを知っておけば理解の助けになる。さまざまなオプション機能を提供しているのは各種のライブラリだ—ユーザーはそれをオプション機能だとは考えないかもしれないが。たとえば、組み込みのオーディオエフェクト以外にAudacityはLADSPA (Linux Audio Developer's Simple Plugin API) をサポートしており、オーディオエフェクトのプラグインを動的に読み込むことができる。AudacityのVAMP APIは、オーディオ解析用のプラグインのために同じ仕組みを用意している。これらのAPIがなければAudacityの機能はあまり豊富とはいえなくなるだろうが、プラグインの機能に依存しないものとなる。

Audacityが使うその他のオプションライブラリにはlibFLACやliboggそしてlibvorbisがある。これらは、さまざまなオーディオ圧縮フォーマットに対応する機能を提供する。MP3フォーマットへの対応は、LAMEあるいはFFmpegライブラリを動的に読み込むことで行う。ライセンス上の制約のため、これら主要圧縮フォーマットのライブラリはAudacity本体に組み込むことができない。

それ以外にも、ライセンスの影響がAudacityのライブラリや構造にあらわれているところがいくつかある。たとえば、VSTプラグインに対応していないのはライセンスの制約のためだ。また、非常に効率的な高速フーリエ変換ライブラリであるFFTWをいくつかの箇所で使っているが、これはAudacityをソースからコンパイルする人でないと使えない。バイナリ版では、多少速度の落ちる別の実装を使うことになっている。Audacityがプラグインの仕組みを受け付ける限り、AudacityではFFTWを使えない。FFTWの作者は、自分のコードを一般的なサービスとして任意のコードで使わせるということを望んでいないのだ。つまり、プラグインをサポートすると決断することで、FFTWを使えないという代償を払うことになる。プラグインをサポートしたおかげでLADSPAプラグインを使えるようになったが、ビルド済みのバイナリ版ではFFTWを使えなくなった。

アーキテクチャを決める際には、開発者たちの限られた時間をいかにして最大限に生かすかということも検討材料となる。我々開発チームの規模は小さいので、使えるリソースも限られている。たとえばFirefoxやThunderbirdの開発チームがセキュリティホールを詳細に調べるのと同じようなことはできない。しかし我々は、Audacityにファイヤウォールを回避するような抜け道を仕込むつもりはない。そこで、AudacityではTCP/IPコネクションを一切扱わないことに決めた。TCP/IPを切り捨てれば、セキュリティに関して考えるべきことを大幅に減らせる。リソースが限られているのを認識したことで、よりよい設計に向かうことが

できた。開発者に必要以上の時間をとらせる機能をカットし、より本質的な機能に集中できるようにしたのだ。

同じく開発者たちの時間を考慮したのが、スクリプト言語の扱いだ。スクリプトで Audacity を操作できるようにしたいが、スクリプト言語を実装するコードは Audacity の中に組み込む必要はない。各言語の処理系を Audacity に組み込んでコンパイルし、ユーザーに好きなものを使わせるというのはあまり意味がない。[1] そのかわりに、スクリプトによる操作はひとつのプラグインモジュールとパイプを使って実装することにした。後ほど説明する。

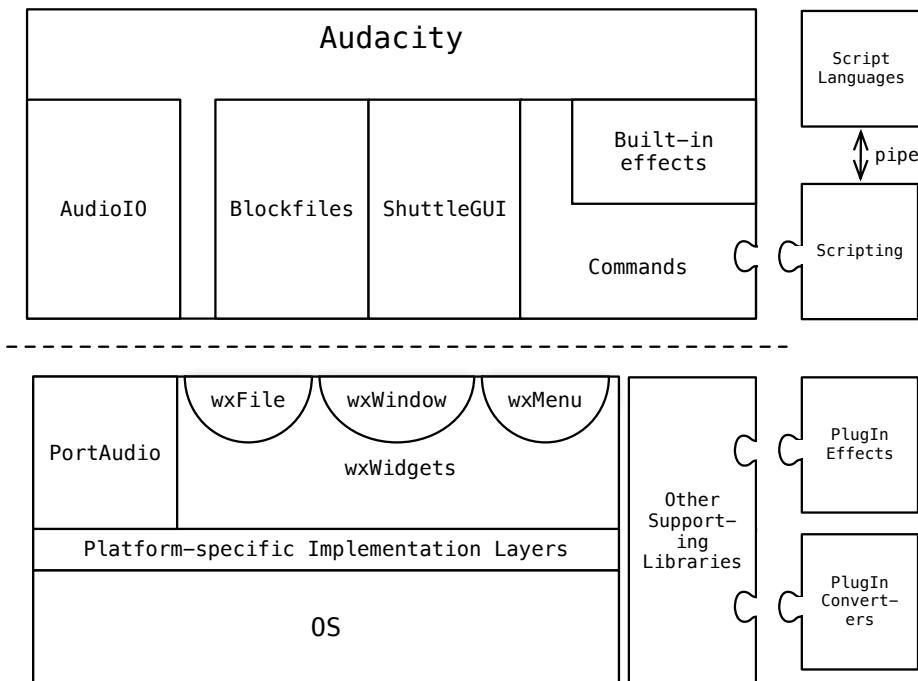



図 2.1: Audacity の階層

図 2.1 は、Audacity のレイヤーとモジュールを図示したものだ。図を見ると 3 つの重要なクラスが wxWidgets 内で強調されており、それぞれに対応するものが Audacity にも用意されている。高レベルの抽象化を、例レベルのクラスに対して行っているのだ。たとえば BlockFile システムは wxWidgets の wxFiles に対応するもので、このクラス上に構築されている。おそらく、いずれは BlockFiles や ShuttleGUI そしてコマンド処理を仲介ライブラリとして切り出すことになるだろう。そうすれば、より汎用的にできる。

図の下のほうには “Platform Specific Implementation Layers” という細長い部分がある。wxWidgets や PortAudio は、どちらも OS を抽象化するレイヤーだ。どちらにも、対象プラットフォー

[1] 唯一の例外は Lisp ベースの言語 Nyquist で、これは Audacity の開発が始まったばかりのころから組み込まれている。できることなら別のモジュールに分割して Audacity にバンドルする形式にしたいところだが、その作業に時間を割く余裕がない。

ムに依存するさまざまな実装にあわせて処理を切り替えるコードが含まれている。

“Other Supporting Libraries” のカテゴリに含まれるのは、さまざまなライブラリの集まりだ。興味深いことに、これらの多くは動的に読み込まれたモジュールを信頼している。これらの動的モジュールは wxWidgets について何も知らない。

Windows プラットフォーム上では、Audacity を単一の一枚岩な実行ファイルとしてコンパイルしていた。つまり wxWidgets と Audacity アプリケーションが同じ実行ファイルに含まれている状態だ。2008 年にこれをモジュラー構造に変更し、wxWidgets は個別の DLL として用意するようにした。これで、さらに別の DLL を実行時に読み込んだときにもそれらの DLL が直接 wxWidgets の機能を使えるようになった。図中の点線より上に組み込むプラグインは、wxWidgets を使うことができる。

wxWidgets を DLL 化したことによるマイナス面もある。まず、配布ファイルのサイズが大きくなった。その理由のひとつは、使ってもいない多くの関数が DLL に組み込まれているからである。DLL 化する前は、これらは最適化されていた。また、Audacity の起動にやや時間がかかるようになった。各 DLL を個別に読み込むためである。しかし、メリットも多い。モジュール化することで、ちょうど Apache のモジュールと同じようなメリットが得られることを期待している。モジュール化したおかげで Apache 本体は非常に安定するし、その一方で実験的な開発や特殊な機能、新たなアイデアなどはモジュールで試すことができる。モジュールのおかげで、プロジェクトをフォークして別の道を行くという衝動に対抗することができる。モジュール化するという決断は、我々にとって非常に重要なアーキテクチャ変更だったと考えている。これらのメリットが得られるだろうと期待してはいるが、今のところはまだ得られていない。wxWidgets の機能を公開することは単なるはじめの一歩に過ぎず、我々はより柔軟なモジュラーシステムに向かってさらに進んでいく。

Audacity のようなプログラムの構造は、前もって明確に設計できるものではない。時間をかけて成長させていくものだ。全体的に見て、現状のアーキテクチャはうまくいっている。ソースファイルの多くの部分に影響する新たな機能を追加するときには、アーキテクチャと対決している気分になる。たとえば、Audacity は現在ステレオとモノラルのトラックをそれぞれ専用の方法で処理している。Audacity を改造してサラウンドサウンドを処理させるようにしようとすれば、Audacity 内の多くのクラスに手を入れる必要がある。

ステレオのその先に: `GetLink` の物語

Audacity は、これまでチャンネル数を抽象化したことがなかった。そのかわりに、音声チャンネルに結びつけた抽象化を使っている。`GetLink` という関数があり、この関数は、2 チャンネルのときはペアのもう一方の音声チャンネルを返し、モノラルのときには NULL を返す。`GetLink` を使うコードは、まるで最初はモノラル用に書いていたものに後から `(GetLink() != NULL)` を使ってステレオ処理のコードを継ぎ足したように見える。本当にそうだったのかは不明だが、私はきっとそうだろうと踏んでいる。`GetLink` を使って連結リスト内のすべてのチャンネルを反復処理させるといったループはなく、描画やミキシング、そして読み書きのすべてで「ステレオの場合は…」という場合分けが含まれている。もともとのコードが任意の n チャンネルを処理できて、ほとんどの場合は n が 1 か 2 になる、というようにはなっていないのだ。より汎用的なコードにしようとおもったら、約 100 か所にある `GetLink` 関数の呼び出しに手を入れなければならない。ファイル数にして少なくとも 26 以上になる。

`GetLink` を呼び出しているところを検索して適切に変更するのはそんなに複雑な作業ではない。この “問題” を修正するのは、最初に感じたほど大がかりな作業ではないだろう。`GetLink` の物語は、修正が困難な構造的欠陥に関するものではない。それよりもむしろ、比較的小規模な欠陥がいかにコードを蝕んでいくかを示すものだ。

今になってみれば、`GetLink` 関数は private にしてそのかわりにイテレータを提供しておいたほうがよかったのだろう。そうすれば、あるトラック内のすべてのチャンネルを順に処理することができる。これでステレオの場合の処理を特別に書くことも防げるし、音声チャンネルのリストを使うコードはリストの実装を知らなくても書けるようになる。

設計のモジュール化を進めると、内部構造をうまく隠蔽する方向に我々を導いてくれるだろう。外部向けの API を定義して拡張することで、アプリケーションが提供する機能について、よりしっかり見つめなければならないことになる。それによって我々は、外部向け API にとらわれない抽象化を心がけるようになるだろう。

2.2 wxWidgets GUI ライブラリ

Audacity のユーザーインターフェイスのプログラマーにとって一番重要なライブラリをひとつあげるとすれば、それは wxWidgets GUI ライブラリだ。このライブラリは、ボタンやスライダ、チェックボックス、ウィンドウそしてダイアログなどを提供する。つまり、目に見えるクロスプラットフォームな挙動の大半をこのライブラリが提供していることになる。

wxWidgets ライブラリには自前の文字列クラス wxString が用意されており、スレッドやファイルシステムそしてフォントなどをクロスプラットフォームで抽象化する。また他の言語へのローカライズにも対応しており、これらすべての機能を Audacity で使っている。Audacity の開発に新たに参加しようとする人に勧めるのは、まず wxWidgets をダウンロードしてコンパイルし、付属のサンプルをいくつか試してみることだ。wxWidgets は、OS が提供する GUI オブジェクトの上に乗る比較的薄めのレイヤーである。

複雑なダイアログを組み立てるために、wxWidgets では個々のウィジェットの要素だけでなくサイザー (sizer) も用意している。これは要素のサイズや位置を制御するもので、図形要素に対して固定の座標での位置指定をするよりもずっとよい。ユーザーが直接ウィジェットのサイズ変更したり、フォントサイズを変更したりしても、ダイアログ内の要素の位置は自然に更新される。サイザーは、クロスプラットフォームなアプリケーションにとって重要となる。もしこれがなければ、ダイアログのカスタムレイアウトをプラットフォームごとに用意しなければならなくなるだろう。

ダイアログのデザインをリソースファイルに書き出すこともよくある。このリソースファイルをプログラムから読み込んで使うのだ。しかし Audacity では、ダイアログの設計は wxWidgets の関数呼び出しの形でプログラムに埋め込んでいる。これによって最大限の柔軟性を確保している。つまり、ダイアログの正確な内容や振る舞いをアプリケーションレベルのコードで決定できるということだ。

かつては、Audacity の GUI を作成するコードの中に、グラフィカルなダイアログ作成ツールで自動生成したのが明らかなコードを見かけることもあった。この手のツールは基本的なデザインを作るときには便利だった。時を経てそのコードにも手が入り、新たな機能が追加された。その後新たなダイアログを作るときには、既存のコードをコピーして流用することが多くなった。すでに自動生成結果に手が加えられたダイアログのコードをだ。

そんな開発を何年も続けてきた結果、Audacity のソースコードの多く (特に、ユーザーの環境設定用のダイアログまわり) に複雑なコードの重複が目立つようになった。かつてはシンプルだったのかもしれないが、そんなコードを追いかけるのは大変だ。問題のひとつは、ダイアログを組み立てる手順がばらばらだったことだ。小さめの要素を組み合わせて大きな要素を作り、最終的にダイアログが完成するようになっていたが、コードで要素を作成する順番は画面上での要素の配置順と一致していなかった (一致させる必要もなかった)。コードは冗長で、繰り返しも多かった。GUI 関連のコードの中には環境設定のデータをディスクから読み出して中間変数に送るものもあったし、中間変数から表示用 GUI に送るコードもあった。また、GUI から中間変数にデータを送るコードもあれば、中間変数からディスクに保存するコードもあった。そのコードのそばには「//これはひどい」とかいうコメントが入っていたが、何か手を付けようとするまでにはかなり時間がかかってしまった。

2.3 ShuttleGui レイヤー

これらのコードのもつれを解決する手段が新たなクラス ShuttleGui だ。これはダイアログの作成に必要なコードの行数を大幅に減らし、コードをより読みやすくする。ShuttleGui は新たなレイヤーで、wxWidgets ライブラリと Audacity との間に挟まるものだ。その役割は、wxWidgets ライブラリと Audacity との間での情報の受け渡しである。例を示そう。これは図 2.2 のような GUI 要素を作成する。

```
ShuttleGui S;
// GUI Structure
S.StartStatic("Some Title",...);
{
    S.AddButton("Some Button",...);
    S.TieCheckbox("Some Checkbox",...);
}
S.EndStatic();
```

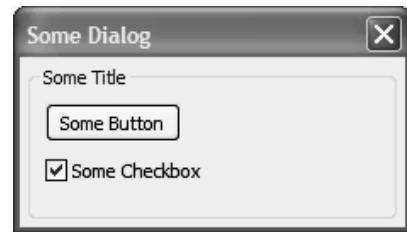



図 2.2: ダイアログの例

このコードはダイアログ内に静的なボックスを定義し、その中にボタンとチェックボックスをひとつずつ置いている。コードとダイアログの対応は明確だ。`StartStatic` と `EndStatic` の呼び出しはペアになっている。それ以外にも同様な `StartSomething/EndSomething` のペアがあり、これらは必ずセットで存在しなければならない。これが、ダイアログ上の配置を制御する。波括弧での囲みやその中の字下げは単にコードの見た目だけにかかわるものであり、必須というわけではない。しかし我々は、このように書くよう規約を定めている。コードの構造や `StartSomething/EndSomething` のペアを明確にするためである。コードが巨大になってくると、これが可読性の向上にとても役立つ。

先ほど示したソースコードは、単にダイアログを作るだけではない。コメント "`//GUI Structure`" の後に続くコードを使って、ダイアログのデータを設定保存先に送ったり逆にデータを取得したりすることができる。これまでは、同様のことをするためには、同じようなコードを大量に繰り返さねばならなかった。今では、コードを一度だけ書けばあとは `ShuttleGui` クラスがうまくやってくれる。

それ以外にも Audacity では、wxWidgets の基本機能を拡張するモジュールを使っている。たとえば、Audacity にはツールバーを管理するための自前のクラスがある。なぜ wxWidget の組み込みのツールバークラスを使わなかったのかって?それは、歴史的な理由によるものだ。Audacity のツールバーが書かれたのは、wxWidgets にツールバークラスが用意されるよりも前のことだったのだ。

2.4 TrackPanel

Audacity のメインパネルで波形を表示しているのが TrackPanel で、これは Audacity のカスタムコントロールだ。このコントロールは、いくつかの小さな部品を組み合わせて作られている。トラック情報を表示するパネルや時間軸のルーラー、振幅のルーラー、そしてトラックの波形やテキストラベルなどだ。トラックのサイズ変更や移動は、マウスのドラッグで行える。トラックにはテキストラベルが含まれているが、これは編集可能なテキストボックスを自前で再実装したものであり、組み込みのテキストボックスではない。これらのパネルやトラック、ルーラーは wxWidgets のコンポーネントにすべきだと考える人もいるだろう。しかしそのようにはなっていない。

図 2.3: Audacity のインターフェイスに、Track Panel の要素名を示したもの

スクリーンショット図 2.3 に Audacity のユーザーインターフェイスを示す。名前がつけられているコンポーネントはすべて、Audacity 用にカスタマイズしたものである。wxWidgets の観点から見れば、ここにある wxWidget のコンポーネントは TrackPanel 用のひとつだけだ。その内部の配置や再描画はすべて、wxWidget ではなく Audacity 側のコードが面倒を見ている。

これらのコンポーネントをうまく取りまとめて TrackPanel を作るのは、本当に恐ろしいことだ (恐ろしいのはあくまでもコードのことであって、実際にユーザーが使う完成品はよくできているけどね)。GUI とアプリケーションのコードが入り混じっていて、きれいに分離でき

ていない。きちんと設計するなら、アプリケーションのコードだけが左右のオーディオチャンネルやそのデシベル、ミュート、ソロなどの設定を関知するべきだ。GUI の要素がそれらを関知しているようではいけない。GUI の要素はオーディオ関連以外のアプリケーションでも再利用可能でなければならない。TrackPanel 上の部品は、純粋な GUI でさえもつぎはぎだらけのコードになっている。絶対位置やサイズによる場合分けが含まれており、十分に抽象化できていない。これらのコンポーネントが自身で GUI 要素を内包して wxWidgets の sizer のようなインターフェイスを使っていれば、どんなにきれいで一貫性のあるコードになるだろう。

TrackPanel をそんなふうに改良するために、トラックやその他のウィジェットの移動やサイズ変更を行う wxWidgets 用の新たな sizer が必要となった。wxWidgets の sizer は、それを満たすほど柔軟ではない。新たに sizer を作ったおかげで、それを他の部分でも使えるようになった。我々はツールバー上に保持するボタンにもこれを使い、ツールバー上のボタンの並べ替えをドラッグで簡単にできるようにした。

新たな sizer を作るために事前調査を行ったが、調査が足りなかったようだ。GUI コンポーネントを wxWidgets から完全に独立させようとする試みたが、問題が発生した。ウィジェットの再描画がうまく制御できなくなり、コンポーネントのサイズを変更したり移動させたりするとちらつきが発生するようになったのだ。wxWidgets をベースにして拡張することで再描画時のちらつきを解決し、サイズ変更の処理と再描画の処理をうまく分離させる必要があった。

TrackPanel をこの方式で改良するのを躊躇したもうひとつの理由は、ウィジェット数が増えると wxWidgets の起動に時間がかかることを既に我々が知っていたからだ。これは、wxWidgets からはあまり手の施しようのない問題だ。個々の wxWidget やボタン、テキスト入力ボックスはそれぞれウィンドウシステムのリソースを使っている。そして、個々のリソースは、アクセスするためのハンドルを持っている。多数のハンドルを処理するのには時間がかかる。たとえ大半のウィジェットが非表示あるいはスクリーンの外部にある場合であっても処理が遅くなることは変わらない。我々は、小さなウィジェットを大量に使うつもりだったのだ。

最もよい解決策は Flyweight パターンを採用することだ。軽量なウィジェットが自分自身の描画を担当し、対応するオブジェクトを持たないようにすれば、ウィンドウシステムのリソースやハンドルを消費せずに済む。我々は wxWidgets の sizer やコンポーネントウィジェットと同様の構造を使い、同様の API を提供するようにした。しかしそれは wxWidgets のクラス群を派生させたものではない。我々は既存の TrackPanel のコードをリファクタリングでよりきれいな構造にした。もしこれが簡単な解決法なら既にそうしていただろうが、自分たちがいったい何を求めているのかについての議論が発散した結果、初期の試みから脱線してしまった。現在のアドホックな方式を一般化するには、大変な設計作業とコーディングが必要となる。複雑ではあるけれども今きちんと動いているコードをそのまま残しておきたいという強い誘惑にもかられた。

2.5　PortAudio ライブラリ: 録音と再生

PortAudio はオーディオライブラリで、Audacity はこれを使って、録音と再生をクロスプラットフォームな方法で提供している。このライブラリがなければ、Audacity は実行環境のサウンドカードを使うことができないだろう。PortAudio が提供する機能にはリングバッファや録音・再生時のサンプルレート変換などがある。重要なのは、その API によって Mac や Linux そして Windows におけるオーディオ処理の差異を隠ぺいできるということだ。PortAudio の内部には、各プラットフォームで API に対応するための実装ファイルが別々に用意されている。

私はこれまで、PortAudio の内部に立ち入って何をしているのか追いかける必要などなかった。しかし、Audacity と PortAudio の間でどのようなやりとりをしているかを知っておくと便利だ。Audacity は、PortAudio からデータのパケットを受信 (録音) したり、逆にパケットを PortAudio に送信 (再生) したりする。送信や受信が実際のところどのように行われているのか、そしてそれがディスクへの読み書きや描画の更新とどのようにつながっているのか。そのあたりは見る価値があるだろう。

いくつかの異なる処理が、同時に発生する。頻繁に発生し、少量のデータをやりとりし、高速な反応を要するものがあれば、あまり頻繁には発生しないが大量のデータをやりとりしなければならないものもある。こちらについては処理がいつ発生するかはそれほど重要ではない。ここに、処理の内容とその対応に使うバッファとの間のインピーダンスミスマッチが発生する。もうひとつ見る価値があるのは、オーディオデバイスやハードディスクそして表示画面などを扱う部分だ。末端まで必死になって追いかけるつもりはなく、与えられた API を使って作業をすることになる。各プロセスを同じように見て、たとえばすべて wxThread から立ち上げるようにしたいものだが、そのような贅沢はできない (図 2.4)。

ひとつのオーディオスレッドを PortAudio のコードが立ち上げ、それが直接オーディオデバイスとやりとりする。これは、録音や再生を行うものだ。このスレッドは反応が速いものでなければならず、そうでないとパケットを失ってしまう。PortAudio のコードの配下にあるスレッドが `audacityAudioCallback` を呼び、録音時には、新たに受け取った小さなパケットを大きめ (5 秒) のキャプチャバッファに追加する。再生時には、5 秒間の再生バッファから小さな塊を取り出す。PortAudio ライブラリは wxWidgets については一切知らない。そのため、PortAudio が作ったこのスレッドは pthread である。

第二のスレッドを立ち上げるのは、Audacity の AudioIO クラスだ。録音の際に、AudioIO がデータをキャプチャバッファから受け取り、それを Audacity のトラックに追加して表示させる。さらに、十分な量のデータが追加された時点で、AudioIO がデータをディスクに書き込む。このスレッドは、再生時のディスクからの読み込みも行う。ここで鍵となるのが関数 `AudioIO::FillBuffers` といくつかの Boolean 変数の設定で、録音と再生をこのひとつの関数で行う。重要なのは、このひとつの関数で双方向の処理をしているということだ。録音部と再生部を同時に使うことがある。“software play through” で、以前に録音された内容に対する多重録音を行うときだ。AudioIO のスレッド内では、完全に OS のディスク IO にしばられた状態となり、ディスクの読み書きでしばらく待たされることもあるかもしれない。これら

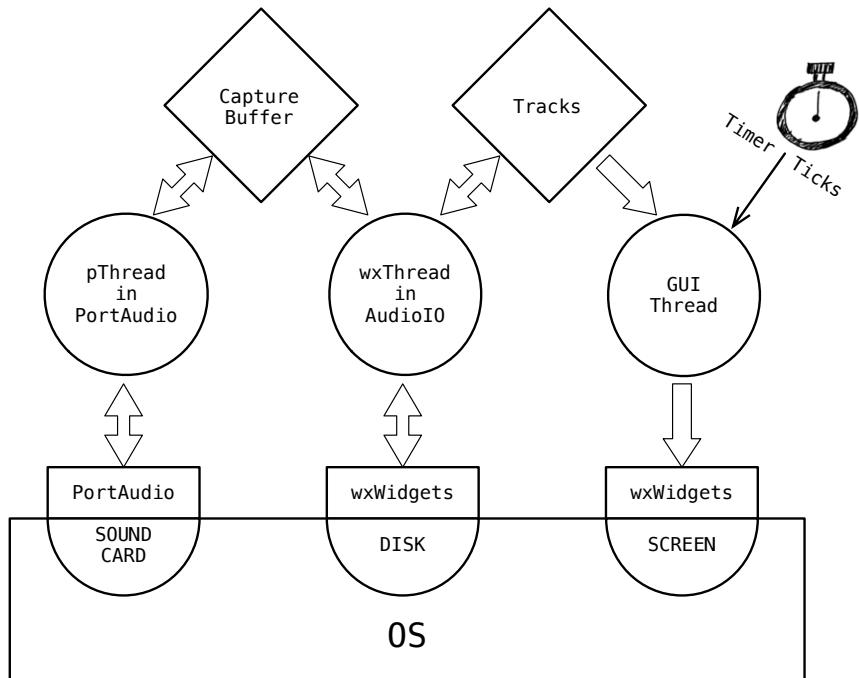



図 2.4: 録音・再生時のスレッドとバッファ

の読み書きを audacityAudioCallback で行うことはできない。この関数は高速な反応を要求されるからである。

これらふたつのスレッド間の通信は、共有変数を使って行う。どちらの変数がいつ書き込みを行っているのかを制御できているので、ミューテックスを使うようなぜいたくは不要だ。

再生と録音の両方で、さらにもうひとつの要件がある。Audacity は GUI も更新しなければならないということだ。これは、最もタイムクリティカルでない処理である。描画の更新はメインの GUI スレッドで行われ、1 秒間に 20 回発生する定期的なタイマーで実行される。このタイマーが `TrackPanel::OnTimer` を呼び出し、GUI の更新が必要な場所が見つかれば更新する。メインの GUI スレッドは、我々のコードではなく wxWidgets の中から立ち上げる。他のスレッドからは、直接 GUI を更新することはできない。タイマーを使って GUI スレッドを取得し、画面の更新が必要かどうかを調べる。そうすることで、再描画の回数を減らしながらも許容できるレベルの反応を保つ。また、そうすることで、表示用にあまりプロセッサタイムを要求しすぎないようにしている。

オーディオデバイス用のスレッドとバッファ/ディスク用のスレッド、そして定期的なタイマーを持つ GUI スレッドの三つを使ってオーディオデータのやりとりをするというのは、よい設計と言えるだろうか?これら三種類のスレッドを単一の抽象基底クラスから派生させないのは、多少アドホックにも感じる。しかし、このアドホック性の要因の多くは、使っているライブラリによるものである。PortAudio は、自分自身でスレッドを作ることを想定している。wxWidgets フレームワークが GUI スレッドを持っているのは、ごく自然なことだ。バッファを埋めるためのスレッドを要する理由は、オーディオデバイスのスレッドで頻繁に発生する小規模のパケットと、あまり頻繁には発生しないディスクドライブの大規模なパケットとの間のインピーダンスミスマッチを解決するためだ。これらのライブラリを使うことには明確な利点がある。逆に、これらのライブラリを使うことで必要となるコストは、結局は各ライブラリが提供する抽象化を使うことになるということだ。結果的に、メモリ内でのデータのコピーの回数が、最低限必要な回数だけでなくさらに増えてしまっている。私がこれまで関わってきた高速なデータ交換の処理ではもっと効率的なコードを見たことがある。そのコードでは、割り込みなどで発生するこの手のインピーダンスミスマッチに対応するのにスレッドなど使っていなかった。データをコピーするのではなく、バッファへのポインタを渡していたのだ。ただ、そんなことができるのは、使っているライブラリがバッファの抽象化をよりリッチに設計している場合だけである。既存のインターフェイスを使う限りはスレッドを使うことを強いられ、そしてデータのコピーを強いられることになる。

2.6 BlockFile

Audacity が直面する困難のひとつが、録音したオーディオデータへのデータの追加や削除だ。オーディオデータの長さは数時間分になることもある。録音データはどんどん長くなり、利用可能な RAM の容量を簡単に突破してしまうだろう。録音データをディスク上に単一のファイルとして管理していたとすると、ファイルの先頭あたりにデータを挿入しようとしたときには大量のデータ移動が発生することになる。ディスク上でのデータのコピーには時間がかかる。つまり、Audacity はちょっとした編集でも長々と待たされるソフトウェアになってしまう。

Audacity がこの問題に対処するために使った方法は、オーディオファイルを多数の BlockFile に分割することだ。個々のファイルの大きさは約 1 MB となる。これが、Audacity が自前のオーディオファイルフォーマットを採用している主な理由だ。マスターファイルの拡張子は .aup となる。これは XML ファイルで、このファイルがさまざまなブロックを取りまとめている。長いオーディオデータの先頭近くを変更した場合でも、影響を受けるのはたったひとつのブロックとマスターファイル .aup だけとなる。

BlockFile は、対立する二つの勢力の調和をうまく保っている。挿入や削除の際に必要以上のコピーが発生することもないし、再生時には、ディスクへのリクエストのたびに適度に大きなデータの塊を取得できることが保証されている。ブロックが小さくなればなるほど、同じ量のオーディオデータを取得するために必要なディスクへのリクエストの回数が増え、逆に大きくなればなるほど挿入や削除の際のコピーの量が増える。

Audacity の BlockFile は決して内部に空き領域を持たないし、最大のブロックサイズを超えることもない。このルールを守るため、挿入や削除をするときには最大 1 ブロックまでのデータのコピーが発生する可能性がある。BlockFile が不要になれば、削除する。BlockFile は参照カウンタで管理されているので、オーディオの一部を削除したとしてもそれに関連する BlockFile はまだ残ったままであり、undo に対応できるようにしている。データを保存するまではその状態になる。Audacity の BlockFile には、空き領域のガベージコレクションはまったく不要だ。我々が対応する必要があるのは、オールインワン型のファイルである。

大きめのデータのマージや分割こそが、データを管理するシステムの本業である。B ツリーから Google の BigTable のタブレットや Unrolled linked list の管理まで、それは変わらない。図 2.5 は、Audacity でオーディオの開始位置付近を削除したときに何が起こるのかを示している。

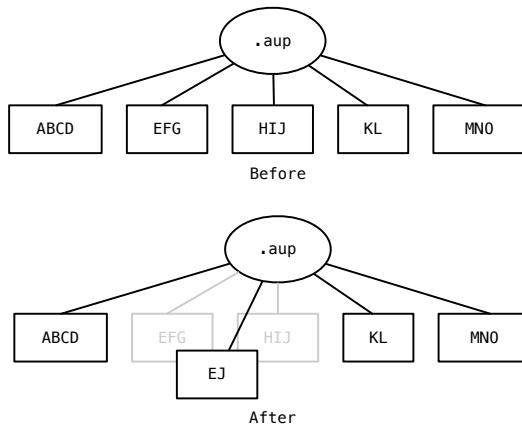



図 2.5: 削除する前は .aup ファイルと BlockFile が保持するのは ABCDEFGHIJKLMNO。FGHI を削除すると、ふたつの BlockFile がマージされる。

BlockFile が扱うのはオーディオそのものだけではない。それ以外にも、サマリ情報をキャッ

シュする BlockFile もある。Audacity で 4 時間のオーディオデータを表示するときに、画面の再描画のたびにオーディオ全体を読み直すのは考えられない話だ。そういう場合にサマリ情報を代わりに使う。サマリ情報には、その時間の範囲内での最大音量や最小音量が記録されている。表示をズームインしたときは、実際のデータを使って描画を行い、ズームアウトしたときはサマリ情報をもとに描画を行う。

BlockFile システムの特徴のひとつに、各ブロックは Audacity が作ったファイルでなくてもかまわないという点がある。たとえば、.wav 形式で保存されたオーディオファイルの特定の時間範囲を指すこともできる。ユーザーが Audacity プロジェクトを立ち上げて.wav ファイルからオーディオをインポートしていくつかのトラックをミックスしたとしよう。このときに作られる BlockFile はサマリ情報を格納するものだけだ。そのおかげでディスク容量も抑えられるし、オーディオデータをコピーする時間も節約できる。しかしながら、これはあまり良い考えではない。これまでに、多くのユーザーがインポート後に元の.wav ファイルを削除してしまっていた。Audacity のプロジェクトフォルダに全部コピーされているものと勘違いしていたのだ。実際はそうではなく、元の.wav ファイルがなければそのプロジェクトは再生できない。そのため、Audacity の現在のデフォルト設定では、インポートしたオーディオを常にコピーして新しい BlockFile を作るようにしている。

BlockFile 方式が、Windows 上で問題となることがあった。Windows 上で大量の BlockFile を扱うと、パフォーマンスが非常に劣化したのだ。原因はおそらく、Windows では同一ディレクトリ上にある大量のファイルの処理速度に難があったからだろう。同様の問題が、ウィジェットをたくさん使ったときの速度低下という形でもあらわれた。その後、サブディレクトリ階層を使うように変更を加え、ひとつのディレクトリには最大 100 個までのファイルしか置かないようにした。

BlockFile の構造を使うときの最大の問題は、その構造がエンドユーザーに公開されてしまうことだ。よく聞く話だが、.aup ファイルを別の場所に移したユーザーが、BlockFile を含むフォルダも一緒に移動させなければならないことに気付かないということがある。Audacity のプロジェクトが単一のファイルにまとまっていて、その内部のスペースやファイル内の利用状況を管理できる状態であればよかっただろう。こうすれば、パフォーマンスはどちらかといえば向上するだろう。追加で必要となるコードは、おそらくガベージコレクションであろう。シンプルなアプローチとしては、ファイルのみ使用領域が設定した割合を上回るときにブロックを新しいファイルへコピーするという方法がある。

2.7 スクリプト

Audacity には、複数のスクリプト言語に対応する実験的なプラグインが付属している。このプラグインは、名前付きパイプを使ったスクリプトインターフェイスを提供する。スクリプト用に公開されたコマンドはテキスト形式で、同じくコマンドの応答もテキストとなる。つまり、名前付きパイプへのテキストの書き出しや名前付きパイプからのテキストの読み込み

に対応しているスクリプト言語ならなんでも、Audacity を動かせるということだ。オーディオそのものなどの大きなデータにパイプ上を行き来させる必要はない (図 2.6)。

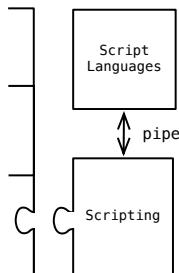



図 2.6: スクリプトプラグインが、名前付きパイプを使ったスクリプト処理機能を提供する

プラグイン自身は、自分が運ぶテキストの内容については何も知らない。単にそれを運ぶだけだ。スクリプトプラグインが使うプラグインインターフェイス (あるいは原始的な拡張ポイント) は、Audacity 側でテキスト形式のコマンドとして公開されている。そのため、スクリプトプラグインは小さなプラグインで、コードの大部分はパイプを扱う処理である。

残念ながら、パイプを使うということは TCP/IP 接続を扱うのと同じセキュリティリスクを負うことになる—セキュリティを考慮して Audacity では TCP/IP 接続を扱わないことに決めている。リスクを少しでも下げるために、このプラグインはオプションの DLL としている。これを取得して使う前には熟考が必要だ。また、このプラグインを使うときにはセキュリティに関する警告が出る。

スクリプトの機能を公開した後で、Wiki の機能追加リクエストのページでこんな提案を受けた。KDE の D-Bus を使えば TCP/IP によるプロセス間通信機能を提供できるのではないか、というものだ。既に別のやりかたで始めたところではあるが、最終的に D-Bus をサポートするようになる可能性も残っている。

2.8 リアルタイムエフェクト

Audacity にはリアルタイムエフェクトの機能はない。再生時にその場で計算してエフェクトをかけるような機能のことだ。Audacity で何かのエフェクトを適用すると、処理が完了するまで待たされることになる。リアルタイムエフェクトを可能にすることやエフェクト処理をバックグラウンドで実行させてその間にもユーザーインターフェイスを機能させることは、Audacity への機能追加要望としてもっともよくあげられるものだ。

問題は、あるマシン上でリアルタイムエフェクトが機能したとしても、別の遅いマシンではとてもリアルタイムとは言えないような速度しか出ない可能性があるということだ。Audacity はさまざまなマシン上で動作する。そのため、穏やかな代替機能を用意しておきたい。多少処理速度の遅いマシンでもエフェクトをトラック全体にかけられるようにし、トラックの中

スクリプト機能のはじまり

スクリプト機能ができたきっかけは、ある Audacity ファンから提供された機能だった。それはあるニーズを満たすための機能で、当初は Audacity をフォークする方向に向かっていた。その機能はひとまとめにして CleanSpeech と呼ばれ、教会での説教を mp3 に変換するために作られた。CleanSpeech には無音部分の切り詰め—オーディオ内の長い無音部分を探して切り取る—などの新たなエフェクトやノイズ除去効果のある固定シーケンスを適用する機能があり、オーディオの正規化や録音内容の mp3 への変換機能などもあった。中にはぜひ取り込みたくなるような素晴らしい機能もあったが、その実装は Audacity 内ではかなり特殊なものだった。それを Audacity の本流に取り込むと、固定シーケンスではなく可変シーケンスのコードを書くことになった。可変シーケンスだと、コマンド名と `Shuttle` クラスのルックアップテーブル経由で任意のエフェクトを使い、コマンドのパラメータはテキスト形式でユーザー設定項目に格納することができた。この機能は**バッチチェイン**と名付けられた。条件分岐や計算式を追加して楽をすることを意図的に避け、アドホックなスクリプト言語を作ってしまわないようにした。

今にして思えば、フォークを避けようと努力をする価値はあった。今でも CleanSpeech モードは Audacity に埋め込まれており、環境設定で有効にすることができる。さらにユーザーインターフェイスも減らし、高度な機能は削除した。シンプルにしたバージョンの Audacity は、他の用途で使いたいという要望もくるようになった。中でも特筆すべきなのは学校での採用だった。問題は、どれが高度な機能でどれが必要不可欠な機能なのかについての意見が人によって異なっていたということだ。我々はその後シンプルなハックを行い、翻訳の仕組みを向上させた。メニュー項目の翻訳のうち "#" で始まるものは、メニューに表示しないようにしたのだ。これで、メニューの項目が多すぎると感じる人も再コンパイルなしでメニューを減らせるようになった—これはより汎用的であり、Audacity の `mCleanspeech` フラグほどには侵略的ではない。このフラグは、いつか取り除いてしまいたいものだ。

CleanSpeech の作業は、我々にバッチチェインと無音切り詰め機能をもたらした。どちらも、コアチーム以外から持ち込まれた魅力的な機能改善だ。バッチチェインはその後のスクリプト機能につながった。それはまた、より汎用的なプラグイン機能を Audacity に持ち込むきっかけにもなった。

央近くまでは処理済みのオーディオを聞けるようにしたい。少しウェイトを入れ、Audacity が最初に処理すべき部分を判断することになる。エフェクトをリアルタイムでレンダリングするには遅すぎるマシンでは、再生がレンダリングに追いつくまではオーディオを聞けるようにしておきたい。そのために必要なのは、オーディオエフェクトがユーザーインターフェ

イスを奪ってしまったり、オーディオブロックの処理を左から右へ順にしなければならなかったりといった制約を取り除くことだ。

比較的最近 Audacity に追加された**オンデマンド読み込み**機能には、リアルタイムエフェクトに必要となる要素の多くが含まれている。しかし、オーディオエフェクトにはまったくかかわっていない。オーディオファイルを Audacity にインポートするときに、サマリ情報の BlockFile の作成は今ではバックグラウンドで行われる。Audacity はプレースホルダーとして青とグレーの斜めの縞をオーディオの部分に表示し、まだ処理が完了していないことを示す。そして、オーディオの読み込み中であっても多くのユーザーコマンドに反応することができる。ブロックの処理を左から右へ順に行う必要はない。このコードは、きっとリアルタイムエフェクトにも使われることになるだろうと確信している。

オンデマンド読み込み機能は、リアルタイムエフェクトの実現に向けた一歩となる。エフェクト自身をリアルタイムで行うことに関する複雑性を、いくらか回避してくれるだろう。リアルタイムエフェクトでは、さらにブロック間のオーバーラップが必要となる。そうしないと、エコーのようなエフェクトが正しくつながらない。また、再生中にオーディオのパラメータを変更することにも対応しなければならない。オンデマンド読み込みを最初に実装したおかげで、他に比べて早い段階からコードを使うことができる。実際の使用例からのフィードバックも得られるだろう。

2.9 まとめ

本章の前半で説明したのは、よりよい構造がいかにプログラムの成長につながるか、そして構造に気を使わないことがいかに開発の妨げになるか、ということだった。

- PortAudio や wxWidgets といったサードパーティの API には多大な利点がある。きちんと動作するコードを組み込めるというだけではなく、プラットフォームの差異をうまく抽象化してくれる。サードパーティの API を使う代償は、抽象化の方法を自由に選ぶという柔軟性がなくなることだ。再生や録音のコードはとても美しいとは言えないものだが、これは三種類の異なる方法でスレッド管理する必要があったからだ。また、このコードにはデータのコピーが多いが、もし抽象化を自由にできていればもう少し減らせたはずだ。
- wxWidgets が我々にもたらす API は、ついつい冗長で追いづらいコードを書きたくなってしまうものだった。その誘惑から逃れるため、我々は wxWidgets の前に Facade を用意して自分たちの求める抽象化をできるようにした。これによって、アプリケーションのコードがよりきれいになった。
- Audacity の TrackPanel では、既存のウィジェットから容易に得られる機能を超えたものが必要となった。その結果、自前のアドホックなシステムを用意することになった。ウィジェットや sizer と論理的に区別されたアプリケーションレベルのオブジェクトからなるよりきれいなシステムが TrackPanel から出てくるように戦っている。

- 構造に関する決定とは、単に新機能をどのように実装するかを決めるだけにとどまらない。プログラムに何を含めないかを決めるのも重要だ。そうすれば、よりきれいで安全なコードにつながる。Perl のようなスクリプト言語の恩恵を受けるときに自分たちのプログラムをいじらなくて済むのはとてもありがたいことだ。構造に関する決定は、将来の成長戦略に基づくものでもある。我々のモジュラーシステムはまだ産まれたばかりのものだが、これを生かしてより多くの実験をより安全に行えることを期待している。また、オンデマンドの読み込み機能は、オンデマンドでのリアルタイムエフェクト処理に進化していくことを期待している。

見れば見るほど明らかなのは、Audacity がコミュニティの尽力の成果だということだ。コミュニティとは、単に Audacity に直接貢献している人たちだけを指すのではない。Audacity はさまざまなライブラリに依存しており、個々のライブラリにはそのコミュニティもあればその分野のドメインエキスパートもいるであろうからだ。Audacity のいろいろ入り混じった構造についての記事を読んだ人にとっては何の驚きもないだろうが、コミュニティは新しい開発者の参入を歓迎しており、スキルレベルに応じていろいろなことをすることができる。

私は、Audacity のコミュニティの質がそのコードの強みや弱みに反映されていると信じて疑わない。より閉じたグループで開発すれば今よりも高品質で一貫性のあるコードが書けるかもしれない。しかし、そんなことをすれば貢献者の数も減り、Audacity が持つ幅広い機能に対応するのは難しくなるだろう。

第3章

The Bourne-Again Shell

Chet Ramey

3.1 導入

Unix のシェルは、ユーザーと OS との間のコマンドによるインターフェイスを提供する。しかし、シェルはまた、リッチなプログラミング言語でもある。フロー制御やループそして条件分岐といった制御構造もあるし、基本的な数学演算や関数、文字列変数などもあり、シェルとコマンドの間の双方向の通信もある。

シェルは、ターミナルあるいはターミナルエミュレータ (xterm など) から対話的に使うこともできるし、コマンドをファイルから読み込むこともできる。bash を含むモダンなシェルにはコマンドラインの編集機能があり、コマンドの入力中に emacs 風あるいは vi 風の操作でコマンドラインをいじることができる。また、さまざまな形式でコマンド履歴を記録する。

Bash の処理はシェルのパイプラインとそっくりだ。ターミナルあるいはスクリプトから読み込んだデータはいくつかのステージを通過し、各ステージで変換され、シェルが最終的にコマンドを実行してその返り値を受け取る。

本章では、bash の主要なコンポーネントである入力処理やパース、さまざまなワードの展開、その他のコマンド処理、そしてコマンドの実行について、パイプラインの観点から探求する。これらのコンポーネントはキーボードやファイルから読み込んだデータのパイプラインとして働き、それを実行されるコマンドに変える。

Bash

Bash は GNU オペレーティングシステムで使われているシェルであり、一般的には Linux カーネル上で実装されている。また、Mac OS X などその他の主要 OS 上でも動く。過去の歴史上のバージョンである sh に対して、対話的な操作においてもプログラミング機能においても改良が施されている。

名前の由来は Bourne-Again SHell の頭文字をとったもので、Stephen Bourne(現在の Unix シェルの先祖である`/bin/sh` の作者。このシェルはベル研の Version 7 Unix で登場した) の名

図 3.1: Bash のコンポーネントのアーキテクチャ

前と再実装によって生まれ変わったことをかけている。bash の最初の作者は Brian Fox で、彼は Free Software Foundation のメンバーだった。私は現在の開発者兼メンテナーであり、オハイオ州クリーブランドにあるケースウエスタンリザーブ大学に勤務している。

他の GNU ソフトウェアと同様、bash も移植性がきわめて高い。Unix のほぼすべてのバージョンで動作するし、その他の OS でも動作する—独自にサポートしている移植版には Windows 上の Cygwin や MinGW といった環境もあるし、QNX や Minix といった Unix ライクなシステムへの移植版は配布物に含まれている。ビルドして実行するために必要なのは Posix 環境だけである。つまり、Microsoft の Services for Unix (SFU) などでもよい。

3.2 構文単位およびプリミティブ

プリミティブ

bash には、基本的に三種類のトークンがある。予約語 (reserved word)、単語 (word)、そして演算子 (operator) だ。予約語とはシェルやそのプログラミング言語に対して何らかの意味

を持つ単語のことで、フロー制御構文に使われることが多い。たとえば`if`や`while`がそれにあたる。演算子とはメタ文字を組み合わせたもののことで、メタ文字とはシェル自身に対して特別な意味を持つ文字を指す。|や>などだ。それ以外のシェルへの入力は普通の単語で、その中にはコマンドライン内での登場位置によって特殊な意味を持つもの—代入文や数値など—もある。

変数およびパラメータ

他のプログラミング言語と同様、シェルにも変数の機能があり、保存したデータを後で参照したり演算に使ったりすることができる。シェルが提供している変数には、ユーザーが設定可能な基本的な変数と、パラメータとして参照できる組み込みの変数がある。シェルのパラメータは一般的にシェルの内部状態を反映するもので、自動的に設定されたり別の操作の副作用として設定されたりする。

変数の値は文字列である。値の中には状況によって特別な意味を持つものもあるが、それについては後で説明する。変数への代入は、`name=value`形式の文を使う。`value`は必須ではなく、省略した場合は空の文字列を`name`に代入する。valueを指定すると、シェルはその内容を展開して`name`に代入する。シェルは、変数が設定されているかどうかによって処理を変えることがある。しかし、変数に値を設定するには値を代入する以外の方法はない。値を代入されていない変数は、たとえ事前に宣言されていたとしても参照すると*unset*となる。

ドル記号で始まる単語は、変数あるいはパラメータへの参照を意味する。ドル記号を含めた単語が、その名前の変数の値に置きかえられる。シェルには豊富な展開演算子が用意されており、単純な値の置換だけではなくパターンにマッチする部分を変更したり削除したりすることもできる。

変数には、ローカルとグローバルの二種類がある。デフォルトでは、すべての変数はグローバルとなる。単純なコマンド (最も見なれた形式のコマンド—コマンド名の後にオプションで引数やリダイレクトが続く形式) の前には代入文がくることもあり、そのコマンドのためだけに変数が存在することになる。シェルはストアドプロシージャやシェル関数を実装しており、それぞれ関数ローカルな変数を持つことができる。

変数には最低限の型をつけることができる。単純な文字列値の変数に加えて、整数値と配列が使える。整数型の変数は数値として扱われる。文字列を代入するとそれを計算式とみなして展開し、計算結果を変数の値として代入する。配列は、インデックス型と連想型のどちらかになる。インデックス型の配列は数値を添字として使い、連想配列は任意の文字列を添字として使う。配列の要素は文字列であり、望むなら整数値として扱うこともできる。配列の要素に別の配列が入ることはない。

Bashは、ハッシュテーブルを使ってシェル変数の格納や取得を行う。また、そのハッシュテーブルの連結リストで変数のスコープを実装する。シェル関数の呼び出し用にさまざまなスコープがあり、コマンドの前にある代入文で設定した変数用のテンポラリスコープもある。代入文の後にシェルの組み込みコマンドが続くときは、シェルは変数の参照の解決順序を覚

えておく必要がある。また、連結したスコープが bash にそれを許可しなければならない。実行のネストレベルによっては、走査するスコープの数が驚くほど多くなることもありえる。

シェルプログラミング言語

単純なシェルコマンド、つまり読者の多くが最も見なれているであろうコマンドは、まず echo や cd のようなコマンド名があってその後にゼロ個以上の引数やリダイレクトが続く。リダイレクトを使うと、起動するコマンドへの入力やコマンドからの出力をシェルのユーザーが制御できるようになる。先ほど説明したように、単純なコマンド内のローカル変数を定義することができる。

予約語を使えば、より複雑なシェルコマンドを実行できる。他の高級言語にもよくある制御構造である if-then-else や while も使えるし、for ループで値のリストを順に処理することもできる。また、C 言語風にカウンタを用いた for ループも使える。これらの複雑なコマンドを使えば、ある条件を調べてその結果によって処理を切り替えるようなコマンドを実行することもできるし、あるコマンドを複数回実行することもできる。

Unix が計算機界にもたらした贈り物のひとつがパイプラインである。これを使えば、一連のコマンド群でひとつのコマンドの出力を次のコマンドへの入力とすることができる。シェルの制御構造はすべてパイプラインの中でも使え、あるコマンドがデータをループに送るようなパイプラインを見ることも珍しくない。

Bash には、あるコマンドの実行時に標準入力や標準出力そして標準エラー出力をリダイレクトして別のファイルやプロセスに送る機能がある。シェルプログラマーは、リダイレクトを使って現在のシェル環境でファイルを開いたり閉じたりすることができる。

Bash では、シェルのプログラムを保存して再利用することができる。シェル関数やシェルスクリプトは、どちらもコマンド群に名前をつけて実行できるようにしたものであり、他のコマンドと同じように実行できる。シェル関数の宣言は特別な構文で行い、同じシェルのコンテキストで使うことができる。シェルスクリプトはコマンドを書いたファイルとして作り、実行するときにはそれを解釈する新たなシェルのインスタンスを立ち上げる。シェル関数は大半の実行時コンテキストを呼び出し元のシェルと共有するが、シェルスクリプトは新たなシェルを立ち上げて動作するので、環境変数で渡された内容しか共有できない。

さらなる注意

さらに読み進めていくうえで覚えておいてほしいのは、シェルがその機能を実装するために使っているデータ構造はほんのわずかであるということだ。配列、ツリー、片方向連結リスト、双方向連結リスト、そしてハッシュテーブル。これだけである。シェルのほぼすべての構造が、これらのプリミティブを用いて実装されている。

あるステージから次のステージに情報を渡したり各処理ステージでデータを操作したりするときに使う基本的なデータ構造が WORD_DESC だ。

```
typedef struct word_desc {
  char *word;           /* Zero terminated string. */
  int flags;            /* Flags associated with this word. */
} WORD_DESC;
```

単語を組み合わせて引数リストなどを作るときには、単純な連結リストを使う。

```
typedef struct word_list {
  struct word_list *next;
  WORD_DESC *word;
} WORD_LIST;
```

WORD_LIST はシェル全体に広がる。単純なコマンドは単語のリストだし、その展開結果も単語のリスト、そして組み込みのコマンドも引数の一覧を単語のリストで受け取る。

3.3 入力の処理

bash のパイプライン処理における最初のステージは、入力の処理である。ターミナルあるいはファイルから文字を受け取り、それを行単位に分け、各行をパーサに渡してコマンドに変換する。想像がつくだろうが、行とは改行文字で終わる文字列のことだ。

Readline およびコマンドラインの編集

Bash は、対話モードのときにはターミナルから入力を読み込み、それ以外の場合は引数で指定したスクリプトファイルから入力を読み込む。対話モードのときは、ユーザーが入力したコマンドラインを編集することができる。編集時には、Unix のエディタ emacs や vi とよく似たキーシーケンスや編集コマンドが使える。

Bash は readline ライブラリを使ってコマンドラインの編集を実装している。このライブラリが提供する関数を使うと、コマンドラインの編集や入力内容の保存、過去のコマンドの呼び出し、そして csh 風の履歴の展開ができるようになる。Readline はもともと bash 用に開発されたものであり、今でも一緒に開発が進められているが、readline には bash 固有のコードは一切含まれていない。多くのプロジェクトが、readline を使ってターミナルベースの行編集インターフェイスを提供している。

Readline には、任意の長さのキーシーケンスを readline コマンドにバインドする機能もある。Readline には、カーソルの移動やテキストの挿入・削除、前の行の取得、そして途中まで入力した単語の補完などに対応するコマンドがある。これらのコマンドを使い、ユーザーはキーバインドと同じ構文でマクロを定義できる。マクロとは、キーシーケンスに対応して挿入される文字列のことである。マクロのおかげで、readline のユーザーはちょっとした文字列置換や作業の短縮をできるようになる。

Readline の構造

Readline は、読み込み/送出/実行/再表示という基本的なループで構成されている。まず最初に、キーボードからの文字の読み込みを read などで行うか、あるいはマクロからの入力を取得する。個々の文字は、キーマップ (ディスパッチテーブル) のインデックスとして使われる。キーマップのインデックスは 8 ビットの 1 文字であるが、その要素はさまざまなものになり得る。たとえば、キーマップの要素の文字列を別のキーマップとして解決することもある。このようにして、複数文字のキーシーケンスを実装している。また、beginning-of-line のような readline コマンドとして解決させることもあり、これは、そのコマンドを実行する。self-insert コマンドにバインドされた文字は、編集バッファに書き込まれる。あるキーシーケンスをひとつのコマンドにバインドすると同時に、そのシーケンスの一部を別のコマンドにバインドすることもできる (これは、比較的最近追加された機能である)。キーマップに特別なインデックスを追加して、これを実現している。キーシーケンスをマクロにバインドすることで、任意の文字列をコマンドラインに追加することから複雑な編集シーケンスのショートカットを作ることまで大きな柔軟性を実現した。Readline は self-insert にバインドされた文字を編集バッファに格納する。表示するときには、これが画面の難行かを占めることがある。

Readline が管理する文字バッファや文字列は C の char だけによるものであり、必要に応じてそこからマルチバイト文字を組み立てる。内部的には wchar_t は使っていない。速度や記憶容量を考慮したことも理由のひとつだが、編集のコードが書かれた頃にはまだマルチバイト文字のサポートがそれほど広まっていなかったという理由もある。マルチバイト文字をサポートするロケールでは、readline が自動的にマルチバイト文字全体を読み込んで編集バッファに追加する。マルチバイト文字を編集コマンドとしてバインドすることも可能だが、バインドするときにはキーシーケンスとして指定しなければならない。可能ではあるが難しいことであり、通常はそんなことをしようとは思わないだろう。たとえば emacs や vi のコマンドでもマルチバイト文字は使われていない。

キーシーケンスが最終的に編集コマンドに解決されたら、readline はターミナルの表示を更新して結果を反映させる。コマンドの結果が文字をバッファに文字を入れるものであったとしても編集位置を移動させるものであったとしても、あるいは行の一部あるいは全体を書き換えるものであったとしても、これは同様に発生する。バインド可能な編集コマンドの中には、履歴ファイルの編集などのように編集バッファには何も変更を加えないものもある。

ターミナルの表示内容の更新は、一見シンプルなようだが実はかなり複雑だ。Readline は三つの内容を気にかけねばならない。画面に表示されている文字バッファの現在の状態、表示バッファの更新後の内容、そして実際に表示されている文字だ。マルチバイト文字があるため、表示されている文字はバッファの内容と正確に一致するとは限らず、再表示エンジンはそのことを考慮しなければならない。再表示するときに readline は、現在の表示バッファの内容と更新されたバッファを比較して差分を算出し、更新後のバッファを表示に反映させるのに最適な方法を決めなければならない。この問題には長年悩まされてきた (**文字列から**

文字列への修正問題)。Readline は次のようにしている。まずバッファの異なる部分の最初と最後の位置を見つけ、その部分だけを更新してカーソルを前後に移動させるコストを算出し(例: ターミナルのコマンドを発行して文字を消してから新しい文字を追加するのと、単に現在の画面表示を上書きしてしまうのとどちらが効率的か?)、もっともコストの低い方法で更新し、必要に応じて最終行の残りの文字を削除してカーソルを正しい位置に移動させる。

再表示エンジンは、readline の中で間違いなく最も頻繁に変更が入っている部分であろう。変更の大半は機能追加である—最も重大なのは、プロンプト内で非表示文字 (色の変更など) を扱える機能やマルチバイト文字の対応だ。

Readline は編集バッファの中身を呼び出し元のアプリケーションに返す。そしてアプリケーションが、おそらく変更されているであろう結果を履歴リストに保存する。

アプリケーション側からの **Readline** の拡張

Readline がユーザーに対してその振る舞いをカスタマイズするさまざまな手段を提供しているのと同様に、アプリケーションに対してもその機能群を拡張する仕組みをいくつか用意している。まず、バインド可能な readline の関数は標準の引数を受け取ることができ、指定した結果を返すことができる。これを使えば、アプリケーション側で readline を拡張してそのアプリケーションに合わせた関数を作りやすくなる。たとえば bash では 30 以上のバインドコマンドを追加しており、bash 固有の単語補完からシェルの組み込みコマンドへのインターフェイスまでさまざまなものを用意している。

アプリケーションから readline の振る舞いを変更する二番目の方法は、フック関数へのポインタに既知の名前と呼び出しインターフェイスを使うことだ。アプリケーションが readline の内部動作の一部を置き換え、readline の前に割り込み、アプリケーション固有の変換をさせることができる。

非インタラクティブな入力の処理

シェルが readline を使っていない場合は、`stdio` あるいは自前のバッファ入力ルーチンを使って入力を取得する。シェルが対話モードでない場合は、`stdio` よりも bash のバッファ入力パッケージを使うことをお勧めする。なぜなら、Posix は入力の取り込みに奇妙な制約を課すからである。シェルが取り込むのはコマンドのパースに必要な部分だけで、残りはそのまま実行プログラムに渡さなければならない。これは、シェルがスクリプトを標準入力から読み込んでいるときに特に重要となる。シェルは、入力をバッファリングすることを許されている。ただし、ファイルのオフセットをパーサが処理済みの最後の文字の直後にまで戻せる場合に限る。現実的な意味合いで言うと、これはつまり次のような意味である。パイプなどのシーク不能なデバイスからスクリプトを読み込む場合は一文字ずつ読み込まなければならず、ファイルなどから読み込む場合は好きなだけバッファリングできるということだ。

これらの特殊な点を別として、非インタラクティブな入力のシェルでの処理は readline と同様である。つまり、改行文字で区切られる文字列のバッファとして扱う。

マルチバイト文字

マルチバイト文字の処理がシェルに追加されたのは、最初に実装が始まってからかなりの時間がたった後のことだった。この機能は、既存のコードに与える影響を最小限にするように設計された。マルチバイトをサポートしたロケールにいるときは、シェルへの入力はバイト (C の char) のバッファとして格納するが、その中身がマルチバイト文字である可能性も考慮するようになる。Readline は、マルチバイト文字の表示方法を知っている (鍵となるのは、マルチバイト文字が画面上で一文字あたりどの程度の場所をとるかということと、画面に表示するときにバッファから何バイト取り出すべきかということだ) し、前後に移動するときにもバイト単位ではなく文字単位になることなども知っている。それ以外に、マルチバイト文字がシェルの入力処理に影響を及ぼすことはない。シェルのその他の部分については後ほど説明するが、マルチバイト文字を考慮にいれた処理が必要となる。

3.4 パース

パースエンジンが最初にする仕事は字句解析、つまり文字のストリームを単語に区切ってそれに意味を与えるということだ。単語は、パーサが何らかの操作をするときの基本単位となる。単語とはメタ文字で区切られた文字列のことである。メタ文字には、スペースやタブといったシンプルな区切り文字のほかにシェル言語で特殊な意味を持つ文字 (セミコロンやアンパサンドなど) がある。

シェルについての歴史的な問題は、Tom Duff が rc(Plan 9 のシェル) に関するペーパーで述べたとおり、Bourne shell の文法を完全に理解している人が誰もいないということである。Posix シェル委員会は Unix シェルの完全な文法を公開するというすばらしい業績を残した。しかしこの文法には、コンテキストに依存する部分が大量にある。この文法に問題がないわけではない—過去の Bourne shell がエラーなしで許容していた構文のいくつかは許可されていない—が、我々が知る限り最善のものだ。

bash のパーサは Posix の文法の初期版に由来するもので、私の知る限りで唯一の、Yacc あるいは Bison で実装された Bourne シェルパーサである。それ故の困難も存在する—シェルの文法は yacc 形式のパースとはあまり相性がよくなくて、複雑な字句解析を必要とするしパーサと字句解析器との連携も多くなる。

いずれにせよ、字句解析器は入力を readline あるいはその他のソースから受け取り、メタ文字でトークンに切り分け、コンテキストにあわせてトークンを識別し、それをパーサに渡して文やコマンドとして組み立てることになる。多くの部分はコンテキストに依存する—たとえば for という単語は、予約後かもしれないし識別子かもしれない。あるいは代入文や他の単語の一部かもしれない。次の例はコマンドとしてまったく問題のないものである。

```
for for in for; do for=for; done; echo $for
```

これは for と表示する。

ここで、余談としてエイリアスについて説明しよう。Bash では、シンプルなコマンドの先頭の単語をエイリアスで任意のテキストに置き換えられる。エイリアスは完全に単語なので、エイリアスを使えば (あるいは悪用すれば) シェルの文法を変えてしまうこともできる。たとえば、bash が提供していない複合コマンドをエイリアスで実装することもできる。bash のパーサはエイリアスを完全に解析フェーズで実装しているので、パーサは解析器に対してエイリアスの展開が許可されたことを通知しなければならない。

他の多くのプログラミング言語と同様に、シェルでも文字をエスケープして特殊な意味を取り除くことができる。エスケープすれば、&のようなメタ文字をコマンド内で使えるようになる。クォートには三種類の方法があり、クォートしたテキストの扱いがそれぞれ少しずつ異なる。バックスラッシュは、それに続く一文字をエスケープする。シングルクォートは、囲まれた文字をすべてそのまま扱う。ダブルクォートもほぼ同様だが、特定の単語の展開は行う (そしてバックスラッシュの扱いが異なる)。字句解析器は、クォートされた文字や文字列をパーサ側で予約語やメタ文字として扱われないようにする。それ以外に特殊な扱いをするのが$'...'と$"..."だ。前者はバックスラッシュでエスケープされた文字を ANSI C の文字列と同じように展開し、後者は標準の国際化関数を使って文字を翻訳する。前者は幅広く使われているが、後者はあまり使われていない。実際の使いどころがほとんどないからであろう。

パーサと字句解析器の残りのインターフェイスはそれほど難しいものではない。パーサはある程度の量の状態を符号化して解析器と共有し、文法上必要となるコンテキスト依存の解析を行う。たとえば、字句解析器はトークンの型に応じて単語を分類している。(適切なコンテキストにおける) 予約語、通常の単語、代入文などである。これを実現するために、パーサは字句解析器に次のようなことを伝えなければならない。コマンドのパースがどこまで進んだか、複数行の文字列 (“ヒアドキュメント” と呼ばれることもある) を処理しているところかどうか、条件分岐の中にいるかどうか、シェルパターンを展開したものを処理しているのか複合代入文を処理しているのかなどである。

パース段階でのコマンドの置換が終わったことを判断する作業のほとんどは、ひとつの関数 (parse_comsub) にまとめられている。この関数は恐ろしいほどの量になるシェルの構文を知っており、トークン読み込みのコード以上に重複がある。最適化されているとはとても言えない。この関数はヒアドキュメントやシェルのコメントについて知っていなければならないし、それだけでなくメタ文字や単語の区切り、クォート処理、予約語が使えるかどうか (つまり、今 case 文の中にいるのかどうか) なども知っていなければならない。これらを正しく処理できるようになるまでには時間がかかった。

単語の展開の際にコマンド置換を展開するときには、bash はパーサを使って言語構造の終了位置を見つける。文字列を eval 用のコマンドに変換するのに似ているが、この場合は文字列の最後でコマンドが終わるわけではない。これを正しく動作させるには、パーサが右かっ

こをコマンドの終端を認識しなければならない。これは多くの文法導出に例外条件を追加することにつながり、字句解析器は (適切なコンテキストにおける) 右かっこに EOF を表すフラグを立てなければならなくなる。パーサはまた、yyparse を再帰的に起動する前にパーサの状態を保存しておかなければならない。コマンドの置換は、コマンドを読み込む際のプロンプト文字列の展開の一部として発生することもあるからである。入力関数は先読みを実装しているので、この関数は最終的に bash の入力ポインタを正しい位置まで巻き戻さないといけない。入力を文字列やファイルから読み込んでいるときでもターミナルから readline で読み込んでいるときでも同じだ。これが重要なのは、単に入力を読み落とさないようにするためというだけではない。コマンド置換の展開関数が実行用の正しい文字列を組み立てられるようにするためでもある。

同様の問題が、プログラマブルな単語補完でも発生する。これは、あるコマンドのパース中に別の任意のコマンドを実行できるようにするものだ。この問題を解決するために、起動の間にパーサの状態を保存して後で復元している。

クォート処理もまた、非互換性や論争の元となるものだ。Posix シェルの標準規格が最初に発表されてから 20 年がたつが、標準化ワーキンググループのメンバーはいまだにクォート処理の適切な振る舞いについて議論を続けている。先に述べたように、Bourne シェルがその振る舞いの参考にしているのはリファレンス実装だけである。

パーサが返すのはコマンドを表す C の構造体 (ループのような合成コマンドの場合は、その中にさらに別のコマンドが含まれる場合もある) で、それがシェル操作の次のステージ、つまり単語の展開処理に渡される。コマンド構造体は、コマンドオブジェクトおよび単語のリストで構成されている。単語のリストの大半は、コンテキストによってさまざまに変換される。その詳細は次のセクションで説明する。

3.5 単語の展開

パースが終わったら、実行の前に、パース段階で生成された単語の多くを展開することになる。つまり、(たとえば)$OSTYPE を文字列"linux-gnu"に置き換えたりするような処理だ。

パラメータおよび変数の展開

変数の展開は、ユーザーにとってもっともなじみ深いものだ。シェルの変数にはほとんど型付けがなく、わずかな例外を除いて文字列として扱われる。この展開では、パラメータおよび変数の文字列を新たな単語や単語リストに変換する。

展開は、変数の値そのものに対して行われる。プログラマーは、これらを使って変数の値の部分文字列を生成したり値の長さを取得したり、指定したパターンにマッチする部分を先頭あるいは末尾から削除したり、値が指定したパターンにマッチする部分を新しい文字で置き換えたり、アルファベットの大文字小文字を変更したりする。

さらに、変数の状態に依存する展開処理もある。変数に値が設定されているか否かによって、展開や代入の内容が異なってくる。たとえば${parameter:-word}は、もし設定されていればparameterと展開されるが、設定されていなかったり空の文字列が設定されている場合はwordと展開される。

その他いろいろ

Bashはそれ以外にもさまざまな展開を行い、それぞれについて独自の変な規則に従っている。最初に処理されるのはブレースの展開で、これは

```
pre{one,two,three}post
```

のような文字列を次のように展開する。

```
preonepost pretwopost prethreepost
```

コマンドの置換も行われる。これは、シェルの機能であるコマンドの実行と変数の操作をうまく組み合わせたものだ。シェルがコマンドを実行してその結果を収集し、その出力を使って値の展開をする。

コマンド置換の問題のひとつは、コマンドを直接実行してその処理が完了するまで待ち続けるということだ。シェルからコマンドに対して入力を送る簡単な方法はない。Bashでは、プロセス置換という機能を使うことができる。これはコマンド置換とシェルのパイプラインを組み合わせたような機能で、コマンド置換のこれらの欠点を埋め合わせるために使える。コマンド置換と同様にbashがコマンドを実行するが、そのコマンドはバックグラウンドプロセスで動作し、処理が完了するまで待つことはない。この機能の鍵となるのは、bashがコマンドへのパイプを開いて読み書きをしたり、ファイルとして公開して展開の結果を記録したりするという点だ。

次に行われるのはチルダの展開である。当初の意図は、たとえば~alanをAlanのホームディレクトリに変換するというものであった。しかし年月を経てこの機能は成長し、今ではさまざまなディレクトリを指すようになっている。

最後に行われるのが算術式の展開である。$((expression))とすると、expressionの部分をC言語の式と同じルールで評価し、その評価結果を使って展開する。

変数の展開は、シングルクォートとダブルクォートの違いが最も明確にあらわれる処理だ。シングルクォートは一切の展開を禁止する—囲まれた部分は何も変更されずにそのまま展開処理を通過する—が、ダブルクォートの場合はいくつかの展開は許可した上でそれ以外の展開を禁止する。単語の展開やコマンド、算術式、プロセスの置換は行われる—ダブルクォートは、その結果の扱い方にだけ影響を及ぼす—が、ブレースやチルダの展開は行われない。

単語の分割

単語を展開した結果は、シェル変数 IFS 内の文字を区切りとして分割される。これを用いて、シェルはひとつの単語を複数の単語に変換する。$IFS 内のいずれかの文字[1] が結果の中に登場するたびに、bash はそこで単語をふたつに分割する。シングルクォートあるいはダブルクォートで囲まれている場合は、この分割は行われない。

グロブ

結果を分割した後でシェルは、展開された単語をパターンとして解釈し、ファイル名 (ディレクトリパスも含む) とのマッチを試みる。

実装

シェルの基本構造がパイプラインにそったものなら、単語の展開は自分自身に向けた小さなパイプラインとなる。単語の展開における各ステージは、単語を受け取って何らかの変換を施し、それを次の展開ステージに渡す。すべての単語展開が終わったら、コマンドを実行する。

bash の単語展開の実装は、既に説明済みの基本的なデータ構造をもとにしている。パーサが出力した単語群は個別に展開され、その結果の単語が入力となる。WORD_DESC 構造体には、単一の単語展開をカプセル化するために必要な情報をすべて保持できる。flags を使って単語展開ステージに必要な情報を符号化し、情報を次のステージに引き継ぐ。たとえば、展開ステージとコマンド実行ステージでは、パーサがフラグを使って特定の単語がシェルの代入文であることを伝える。また、単語展開のコードでは、このフラグを内部的に使って、単語の分割を禁止したりクォートした null 文字列 ("$x"。ただし$x は未設定あるいは null 値が設定されている) の存在を示したりする。展開される各単語に対して文字列を用意し、何らかの文字符号化で追加情報を表すようになどしていたら、もっと難しくなっていたことだろう。

パーサと同様、単語展開のコードもマルチバイト文字を正しく扱うことができる。たとえば、変数の長さの展開 ($#variable) は、バイト数ではなく文字数を数える。展開のコードは、展開の終わりやマルチバイト文字列で特別な意味を持つ文字を正しく識別する。

3.6 コマンドの実行

bash の内部パイプラインにおけるコマンド実行ステージは、実際のアクションが発生する場所である。ほとんどの場合、展開された単語群はコマンド名と引数群に分けられる。OS に

[1]たいていの場合は、いずれかの文字の列となる。

渡すときには、コマンド名の部分が読み込んで実行するファイルとなり、残りの単語は argv の残りの要素となる。

ここまでの説明は意図的に、Posix が単純なコマンド—コマンド名と引数セットからなるコマンド—を呼ぶ場合に重点を置いている。この手のコマンドが最も一般的であるからそうしたのだが、bash にはそれ以外のコマンドもある。

コマンド実行ステージへの入力は、パーサが組み立てたコマンド構造と、展開された単語群のセットとなる。ここからが、真の bash プログラミング言語の出番だ。このプログラミング言語は先ほど説明したような変数や展開を使うし、高級言語と聞いて一般に思い浮かべるような言語構造を実装している。ループや条件分岐、グルーピング、選択、パターンマッチングによる条件付きの実行、式の評価、そしてシェル特有のいくつかの言語構造などだ。

リダイレクト

OS とのインターフェイスとしてのシェルの役割のひとつを反映しているのが、起動したコマンドの入出力に対するリダイレクト機能である。リダイレクトの構文は、初期のシェル利用者の洗練度を表すものだった。つい最近まで、リダイレクトを使う場合は自分が使っているファイルディスクリプタをきちんと把握しておく必要があったのだ。標準入出力と標準エラー出力だけでなく、それ以外のファイルディスクリプタも番号で明示しなければならなかった。

最近追加されたリダイレクト構文によって、シェルに適切なファイルディスクリプタを選ばせてそれを指定した変数に代入できるようになり、ユーザーがファイルディスクリプタを指定する必要はなくなった。そのおかげでプログラマがファイルディスクリプタを意識する面倒は減らせたが、新たな処理が増えた。シェルがファイルディスクリプタを正しい場所に複製し、指定した変数にそれを代入しなければならなくなった。これは、字句解析器からパーサを通してコマンド実行まで情報を渡していく方法を示すもうひとつの例となる。解析器が変数代入を含むリダイレクトとして単語を識別し、パーサは構文の構築時にリダイレクトオブジェクトを作る。このオブジェクトには代入を要するという意味のフラグを立てる。そして、リダイレクトのコードがそのフラグを読み取り、ファイルディスクリプタの番号を正しい変数に代入する。

リダイレクトの実装で最も難しい部分は、リダイレクトを取り消す方法を覚えておくことだ。シェルは、ファイルシステムから実行して新しいプロセスを立ち上げるコマンドとシェル自身が実行する (組み込みの) コマンドの区別を意図的に曖昧にしている。しかし、コマンドの実装がどうであろうと、リダイレクトの効果をそのコマンドが完了した後までひきずってはいけない[2]。したがって、シェルはリダイレクトの効力を取り消す方法を覚えておかなければならない。さもないと、シェルの組み込みコマンドの出力をリダイレクトしたときにシェルの標準出力が変わってしまう。Bash は、リダイレクトの形式ごとにその取り消し方法を知っている。割り当てられたファイルディスクリプタをクローズするか、あるいは複製さ

[2]組み込みコマンド exec はこのルールの例外だ。

れたファイルディスクリプタを保存してあとで dup2 を使って復元するかのいずれかだ。これらはいずれもパーサが作った同じリダイレクトオブジェクトを使い、同じ関数で処理される。

複数のリダイレクトを単純にオブジェクトのリストで実装しているので、取り消しに使うリダイレクトは別のリストとして保持する。このリストはコマンドが完了したときに処理されるが、シェルはそれがいつ処理されるのかを気にかけねばならない。シェルの関数や組み込みコマンド “.” に関連づけられたリダイレクトは関数や組み込みコマンドが完了するまで有効にしておかなければならないからである。コマンドを起動しないときは、組み込みコマンド exec は取り消し用のリストを破棄する。exec に関連づけられたリダイレクトはシェルの環境に残り続けるからだ。

他にもやっかいなことがあるが、それは bash のせいではない。過去のバージョンの Bourne シェルでは、ユーザーが操作できるファイルディスクリプタが 0 から 9 までだけだった。10 以上は、シェルが内部的に使うために予約されていたのだ。Bash ではこの制限を緩め、プロセスのファイルオープンの上限に達するまで任意のディスクリプタを操作できるようにした。つまり、bash は自身が保持する内部のファイルディスクリプタ (直接シェルからではなく、外部のライブラリからオープンしたディスクリプタも含む) を覚えておいて必要に応じて移動できるようにしなければならないということになる。管理しなければならないことが増えるし、close-on-exec フラグのような仕組みも必要になる。さらに新たなリダイレクトのリストを用意して、コマンドの実行期間や処理済みか破棄されたかなどを管理しなければならない。

組み込みコマンド

Bash には、多数のコマンドがシェル自身の一部として組み込まれている。これらのコマンドはシェルから実行されるもので、新たなプロセスは立ち上げない。

コマンドを組み込みで用意する主な理由は、シェルの内部状態を保ったり変更したりするためだ。cd がよい例である。Unix の授業で最初に行う典型的な課題は、なぜ cd が外部コマンドとして実装できないのかを説明させることだ。

Bash の組み込みコマンドは、シェルのその他の部分と同じ内部プリミティブを使う。組み込みコマンドは C 言語の関数を使って実装されており、この関数は単語のリストを引数として受け取る。単語リストは単語展開ステージの出力する結果であり、組み込みコマンド側ではそれをコマンド名とその引数として解釈する。組み込みコマンドが使う展開ルールはほとんどが他のコマンドと同じ標準的なものだが、いくつか例外がある。bash の組み込みコマンドの中で代入文を引数として受け取れるもの (declare や export など) は、代入の引数を展開するときにはシェルの変数代入のときと同じルールを使う。ここでも WORD_DESC 構造体の flags を使い、シェルの内部パイプライン上でステージからステージへと情報を渡す。

単純なコマンドの実行

単純なコマンドは、最もよく見かける形式のコマンドである。ファイルシステムからのコマンドの読み込みやその実行、そして終了ステータスの取得について説明すれば、ここまで説明してこなかったシェルの機能の多くをカバーすることになる。

シェルの変数への代入 (つまり、`var=value` 形式の単語) は、それ自身が単純なコマンドの一種である。代入文はコマンド名の前に書くこともできるし、コマンドラインでそれ単体で使うこともできる。コマンドの前に書いた場合は、その変数が後に続くコマンドの実行環境内で渡される (組み込みコマンドやシェル関数の前に書いた場合は、いくつかの例外を除いて、その変数が有効なのは組み込みコマンドあるいは関数の実行中だけとなる)。もし代入文のあとにコマンド名を続けなければ、その代入文はシェルの状態を変更する。

指定したコマンド名がシェルの関数や組み込みコマンドにないものであった場合、bash はファイルシステムから、その名前の実行可能なファイルを探す。環境変数 `PATH` の値は、このときの検索先を表すディレクトリをコロン区切りでつなげたリストである。スラッシュ(あるいはそのあのディレクトリ区切り文字) を含むコマンド名は検索対象外となるが、直接実行することはできる。

`PATH` の検索でコマンドが見つかれば、bash はそのコマンド名とフルパス名をハッシュテーブルに保存する。ハッシュテーブルの内容は、その後に `PATH` の検索が発生したときに検索の前に参照される。コマンドが見つからない場合、もし特別な名前の関数が定義されていれば bash はその関数を実行する。コマンド名と引数を、この関数への引数として渡す。Linux のディストリビューションの中には、この機能を使って存在しないコマンドをインストールさせようとするものもある。

実行するファイルが見つかれば、bash はそれをフォークして新しい実行環境を作り、この新しい環境でプログラムを実行する。実行用の環境はシェルの環境を完全に複製したもので、シグナルの処理やリダイレクトでオープンしたりクローズしたりしたファイルなどのちょっとした修正が入っている。

ジョブ制御

シェルによるコマンドの実行には二通りの方法がある。まずはフォアグラウンドでの実行で、これはコマンドが終了するまで待ってその終了ステータスを受け取る。もうひとつはバックグラウンドでの実行で、シェルはすぐに次のコマンドを読み込むことができる。ジョブ制御とは、プロセス (実行されたコマンド) をフォアグラウンドとバックグラウンドの間で移動したり実行の一時停止や再開をしたりする機能のことである。この機能を実装するために bash はジョブという概念を導入した。ジョブとは、基本的にはひとつあるいは複数のプロセスから実行されたコマンドのことである。たとえばパイプラインでは、構成する各要素に対してひとつずつプロセスを使う。プロセスグループを使い、個々のプロセスをひとつにまとめて単一のジョブとする。ターミナルは自身に関連付けられたプロセスグループの ID を持って

いる。つまり、フォアグラウンドプロセスグループとは、ターミナルと同じプロセスグループ ID を持つプロセスグループのことである。

シェルは、いくつかのシンプルなデータ構造を使ってジョブ制御を実装している。子プロセスを表す構造体には、そのプロセス ID や状態、終了時に返すステータスなどが含まれている。パイプラインは、単にこのプロセス構造体をシンプルな連結リストにしただけのものだ。ジョブもまったく同様で、プロセスのリストとジョブの状態 (実行中、停止中、終了など)、そしてジョブのプロセスグループ ID で管理されている。プロセスのリストは、通常は単一のプロセスだけで構成されている。パイプラインの場合だけ、複数のプロセスがジョブに関連付けられる。各ジョブはそれぞれ一意なプロセスグループ ID を持っており、ジョブ内のプロセスの中でプロセスグループ ID と同じプロセス ID を持つものがプロセスグループリーダーと呼ばれる。現在のジョブセットは配列に保持されていおり、この考え方ははユーザーに対する見せ方と非常に似ている。ジョブの状態や終了ステータスは、そのジョブを構成するプロセスの状態や終了ステータスを集約した結果となる。

シェルの他の部分と同様、ジョブ制御を実装するときに複雑になる部分は帳簿管理である。シェルはプロセスを正しいプロセスグループに割り当てねばならないし、子プロセスの作成とプロセスグループへの割り当てを同期させなければならない。またターミナルのプロセスグループも適切に設定しなければならない。ターミナルのプロセスグループがフォアグラウンドジョブを決める (そして、もしシェルのプロセスグループが設定されていなければ、シェル自身がターミナルからの入力を読み込めない) からである。これはとてもプロセス指向な考え方なので、`while` や `for` ループのような複合コマンドを実装してループ全体をひとまとめで開始・停止できるようにするのは難しい。実際、それを実現しているシェルはほとんどない。

複合コマンド

複合コマンドは単純なコマンドのリストで構成されており、`if` や `while` といったキーワードから始まる。複合コマンドのおかげで、シェルのプログラミングの威力を発揮できるようになる。

その実装方法は、特に驚くようなものではない。パーサが複合コマンドに対応するオブジェクトを組み立て、コマンドを解釈するときにはそのオブジェクトを走査していくという方法だ。個々の複合コマンドはそれに対応する C の関数として実装されている。この関数が、適切な展開処理や指定したコマンドの実行、そしてコマンドの返り値に応じた実行フローの切り替えなどを行う。`for` コマンドを実装する関数を例にして説明しよう。この関数は、まず最初に、予約語 `in` に続く単語のリストを展開しなければならない。それから、展開された単語を順にとりあげて適切な変数に代入し、`for` コマンドの本体にあるコマンドのリストを実行することになる。`for` コマンドはコマンドの終了ステータスによって実行を切り替える必要はない。しかし、組み込みコマンド `break` や `continue` の影響には注意する必要がある。

リストにあるすべての単語の処理を終えると、`for` コマンドは処理を返す。これでわかるように、実装の大半はそのコマンドの説明と密接につながっている。

3.7　学んだこと

大切だとわかったこと

私はこれまで 20 年以上 bash にかかわってきて、いくつかのことに気付いた。最も重要なこと—いくら強調してもしすぎではないだろう—は、ChangeLog を詳しく書いておくべきだということだ。あとで ChangeLog を読みなおせば、そのときなぜそんな変更をしたのかを思い出せる。さらに、その変更を特定のバグレポートやそれを再現させるテストケースと結び付けられればすばらしい。

もし可能であれば、回帰テストをプロジェクトの立ち上げ時から組み込んでおくことをお勧めする。Bash には数千のテストケースがあり、非インタラクティブな機能のほぼすべてをカバーしている。インタラクティブな機能についてのテストを組み込むことも検討した—Posix には独自の適合性テストスイートが存在する—が、それに必要なフレームワークを配布することになるのは避けたかった。

標準規格は重要だ。Bash は、標準に従った実装をすることで恩恵を受けている。また、自分が実装しているソフトウェアの標準化作業にかかわることも重要だ。さまざまな機能やその振る舞いについて議論できるだけでなく、規格化するときの参考にする標準を持っていればうまくいく。もちろん、それはうまく動かないかもしれない—規格の内容による。

外部の標準も重要だが、内部的な標準を持つこともまた大切だ。私は幸運なことに GNU Project の標準規格に組み込まれ、設計や実装に関する現実的で優れたアドバイスを得ることができた。

よくできたドキュメントも不可欠だ。もし自作のプログラムを他の人に使わせるつもりなら、包括的で明確なドキュメントを作るだけの価値はある。そのソフトウェアが成功して広まると、いろいろなところで大量のドキュメントが作られることになる。そのときには、開発者自身が書いた正式なバージョンがあることが重要となる。

世の中には、よいソフトウェアが豊富にある。使えるものならどんどん使っていこう。たとえば gnulib には、(gnulib フレームワークから取り出せれば) 便利なライブラリ関数が大量に用意されている。BSD や Mac OS X でも同様だ。ピカソも言っていたように「偉大な芸術家は盗む」のだ。

ユーザーコミュニティは大切にしよう。しかし、ときには批判的な声に悩まされることになるかもしれないことに注意。アクティブなユーザーコミュニティからは非常に多くのメリットを得られるが、その結果として人々が熱くなりすぎることもある。個人的に責められているとは思わないことだ。

私がそれ以外にやったこと

Bash には何百万人ものユーザーがおり、後方互換性の重要性を思い知らされている。ある意味では、後方互換性を保てば決して「ごめんなさい」と言わずに済むともいえる。しかし、世の中はそんなに単純ではない。これまでに、互換性を崩す変更をせざるを得なくなることが何度かあった。そしてそのたびに一部のユーザーから苦情を受けた。しかしそれらの変更は、すべて妥当な理由のあるものだった。間違った決定を正すものだったり、設計時の機能漏れを修正するものだったり、シェルの各パーツ間での非互換性を直すものだったりといった変更だ。初期のうちに、公式な bash 互換性レベル的な何かを出しておけばよかったのだろう。

Bash の開発は、これまでオープンになったことがなかった。私は、マイルストーンリリース (bash-4.2 など) および個別にリリースされるパッチという考え方に満足している。このようにする理由は次の通りだ。私は、フリーソフトウェアやオープンソースの世界よりは長めのリリース間隔を保つベンダーを相手にしている。そして過去に、ベータ版のソフトウェアが思いのほか広まりすぎてトラブルになることがあった。しかし、もし最初からやり直せるのなら、もう少しこまめにリリースしたり何らかの公開リポジトリを用意したりすることも検討するだろう。

そんなことを挙げていったところで、実現性を考慮しなければ完成しない。今までに何度か検討したけれどもできなかったことがひとつある。それは、bash のパーサを再帰下降で書きなおし、`bison` を使わないようにすることだ。そうしないとコマンド置換を Posix 準拠にできないのではないかと考えたのだが、最終的にはそんな大規模な変更をしなくても問題を解決できた。もし bash をスクラッチで書きなおすことになったら、おそらくパーサは自分で書くだろう。そのほうがいろいろ楽になるのは明らかだから。

3.8 結論

Bash は、大規模で複雑なフリーソフトウェアのよい例と言える。20 年を超える開発期間を経て成熟し、かつ強力だ。あらゆる場所で動いており、何百万もの人々が毎日使っている。その多くは bash を使っているということを意識していないだろう。

Bash は多くのソースの影響を受けており、古くは Version 7 Unix のシェル (Stephen Bourne が書いたもの) にまでさかのぼる。最も多くの影響を受けたのは Posix 標準規格で、bash の仕様の大部分は Posix に由来するものだ。過去との互換性を保ちつつ標準に準拠するのは困難なことだった。

GNU Project の一員であることで、Bash は多くの恩恵を受けている。GNU が bash の存在価値を与えてくれたのだ。GNU がなければ bash も存在しなかっただろう。また、bash にはアクティブで活気のあるコミュニティがついている。コミュニティのフィードバックがあったからこそ今日の bash がある—まさにフリーソフトウェアのメリットを体現しているというわけだ。

第4章

Berkeley DB

Margo Seltzer and Keith Bostic

コンウェイの法則によると、ソフトウェアの設計はそれを作った組織の構造を反映したものとなるらしい。もう少し話を広げると、こんなことも言えないだろうか。あるソフトウェアがたった二人によって設計されて作り出されたのなら、単に組織の構造を反映しているだけにとどまらず、二人の好みや思想も持ち込まれているのでは? 二人の片割れである Seltzer は、そのキャリアをずっとファイルシステムやデータベース管理システムの世界に捧げている。彼女はきっとこう言うだろう。「その二つは基本的にまったく同じものでしょう? もっと言うなら、OS とデータベース管理システムだってそう。本質的にはどちらも、リソース管理と便利な抽象化をしているに過ぎないのだから」単に、ちょっとした実装の詳細が違う “だけ” だということだ。もう一方の Bostic はツールベースのアプローチによるソフトウェア開発を信じており、シンプルなブロックを組み合わせて部品を作っていこうとする。そういったシステムの方が常に、モノリシックなアーキテクチャに比べて各種の「○○性」に優れているからである。そう、理解容易性とか拡張性とか保守性とかテスト可能性とか柔軟性といかいうやつだ。

この考えを組み合わせると、私たち二人がこの二十年間の大半を Berkeley DB に費やしてきたと知っても驚かないだろう。高速で柔軟性・信頼性・拡張性に富むデータ管理用のソフトウェアライブラリ、それが Berkeley DB だ。Berkeley DB は、人々がもっと伝統的なシステム、たとえばリレーショナルデータベースなどに期待する機能の多くを提供する。しかし、そのまとめかたは異なっている。たとえば、Berkeley DB はキーによるアクセスもシーケンシャルアクセスも高速に行え、トランザクションにも対応しているし障害回復機能もある。しかし、そういった機能はライブラリとして提供しており、それを使いたいアプリケーション側で直接リンクして使うようになっている。スタンドアロンのサーバーアプリケーションとして提供しているわけではない。

本章では Berkeley DB の詳細を扱う。さまざまなモジュールの集まりで作られていることや、それぞれが Unix の思想である “ひとつのことをうまくやらせる (do one thing well)” を体現していることもわかるだろう。Berkeley DB を組み込んだアプリケーション側からは、こ

れらのコンポーネントを直接使うこともできるし、あるいはもっとシンプルに、慣れ親しんだ get、put、delete といった操作を通じて暗黙的に使うこともできる。ここでは、そのアーキテクチャに注目する。最初はどう考えたのか、どんな設計をしたのか、そして最終的にどうなったのか、それはなぜか、そういったことだ。設計は、状況に合わせて変化することもあり得る (実際、あったし!)。大切なのは、原則を外さずに一貫したビジョンを持ち続けることだ。また本章では、長期間にわたるソフトウェア開発プロジェクトにおけるコードの成長についても簡単に検討する。Berkeley DB はふたつのディケイドにまたがる現在進行中のプロジェクトであり、必然的に「よい設計」にも打撃を与えている。

4.1　はじまり

Berkeley DB が生まれたのは、まだ Unix オペレーティングシステムが AT&T に独占されていた頃のことだ。何百ものユーティリティやライブラリがあったが、厳しいライセンスの制約に縛られていた。Margo Seltzer はその当時カリフォルニア大学バークレー校の大学院生。一方 Keith Bostic はバークレーの Computer Systems Research Group に属していた。当時の Keith が携わっていたのは、AT&T のプロプライエタリなソフトウェアを Berkeley Software Distribution から取り除く作業だった。

Berkeley DB プロジェクトが立ち上がった当時のささやかな目標は、インメモリのハッシュパッケージである `hsearch` やディスク上でのハッシュパッケージである `dbm/ndbm` を置き換えることだった。新しい、そしてより改善されたハッシュ実装を作り、メモリ上でもディスク上でも扱えるようにする。そしてプロプライエタリなライセンスに縛られずに自由に配布できる。それを目指していた。Margo Seltzer が書いた [SY91]`hash` ライブラリは、Litwin の Extensible Linear Hashing に関する研究を利用したものだった。巧妙な方法で一定時間でのハッシュ値とページアドレスのマッピングを実現しただけでなく、ハッシュのバケットやファイルシステムのページサイズ（たいていは 4k バイトや 8k バイト）を超える大きなアイテムも扱えるようにした。

ハッシュテーブルがうまくいくのなら、Btree とハッシュテーブルの組み合わせはもっとうまくいくだろう。同じくカリフォルニア大学バークレー校の大学院生だった Mike Olson はこれまでに数多くの Btree を実装しており、新たにまた実装することにも同意してくれた。我々三人は、Margo の hash ソフトウェアと Mike の Btree ソフトウェアをアクセスメソッドに依存しない API に変換した。アプリケーション側からはデータベースハンドルを通じてハッシュテーブルあるいは Btree を参照でき、データベースハンドルがデータの読み込みや変更をするメソッドを保持していた。

これら二つのアクセスメソッドを元にして、Mike Olson と Margo Seltzer は研究論文を共著した ([SO92])。この論文で論じているのは LIBTP で、これはアプリケーションのアドレス空間で動くプログラムのトランザクションライブラリである。

ハッシュと Btree のライブラリは最終的に 4BSD のリリースに組み込まれ、ここで Berkeley DB 1.85 と名付けられた。技術的には Btree アクセスメソッドが実装しているのは B+link 木

だったが、本章では今後 Btree と呼ぶことにする。アクセスメソッドの名前がそうなっているからだ。Berkeley DB 1.85 の構造や API は、これまでに何らかの Linux や BSD 系システムを使ったことがある人にはなじみやすいものだろう。

Berkeley DB 1.85 ライブラリはその後数年ほとんど動きがなかったが、1996 年に Netscape が Margo Seltzer や Keith Bostic と契約を結び、LIBTP の論文で示された完全にトランザクショナルな設計による商用に耐えるバージョンを作ることになった。その結果できあがったのが、初めてトランザクションをサポートしたバージョンである Berkeley DB 2.0 だ。

それ以降の Berkeley DB の歴史はシンプルで、よくありがちな流れになっている。Berkeley DB 2.0 (1997) で、トランザクションが Berkeley DB に導入された。Berkeley DB 3.0 (1999) は新たに設計しなおしたバージョンで、さらに高度な抽象化によってさまざまな機能の増加に対応した。Berkeley DB 4.0 (2001) ではレプリケーションや高可用性に関する機能が導入され、Oracle Berkeley DB 5.0 (2010) では SQL が使えるようになった。

本章の執筆時点では、Berkeley DB は世界で最も広く使われているデータベースツールキットである。何億ものコピーが、ルーターやブラウザからメールソフトや OS まであらゆるところで動いている。20 年以上たった現在でも Berkeley DB のツールベースでオブジェクト指向なアプローチは現役であり、自身をインクリメンタルに改良して使う側のソフトウェアの要求に応え続けている。

設計講座 **1**

複雑なソフトウェアパッケージをテストしたり保守したりしていく上で必須なのが、ソフトウェアをうまくモジュール分割した設計にしておいてモジュール間をよくできた API で連携させることだ。モジュールの境界はニーズにあわせて移動できる (移動できるべき!) が、常に境界が必須だというわけではない。モジュールの境界があれば、ソフトウェアがメンテナンス不能なスパゲッティ状態になることを防げる。Butler Lampson はかつてこう言った。「計算機科学の世界のあらゆる問題は、別のレベルの手段で解決できる」。さらに、何かがオブジェクト指向であるとはどういうことなのかと聞かれた Lampson の答えは「API を介して複数の実装を持てるということさ」だった。Berkeley DB の設計や実装はこの考え方を具現化したものであり、複数の実装を共通のインターフェイスで扱えるようになっている。オブジェクト指向の見た目を持っているが、実際のところこのライブラリは C で書かれているのだ。

4.2 アーキテクチャの概要

この節では Berkeley DB ライブラリのアーキテクチャを取り上げる。まずは LIBTP から始め、そしてその進化の鍵となる局面を強調する。

図 4.1 は Seltzer と Olson の最初の論文から引用したもので、当初の LIBTP のアーキテクチャを示している。一方図 4.2 は、Berkeley DB 2.0 で計画していたアーキテクチャである。

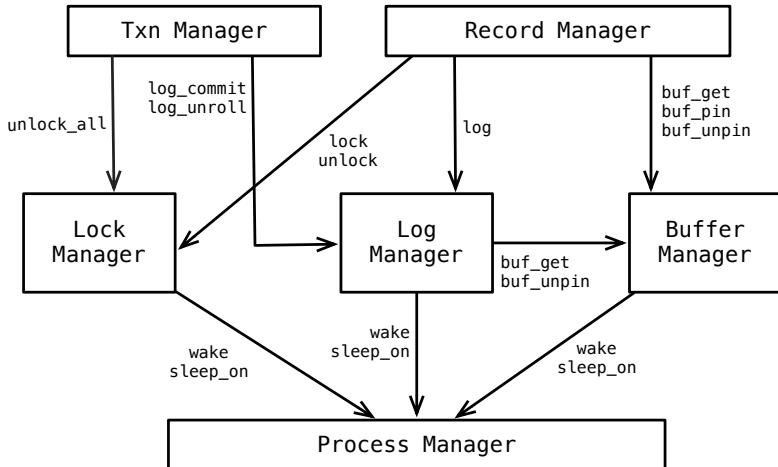



図 4.1: LIBTP プロトタイプシステムのアーキテクチャ

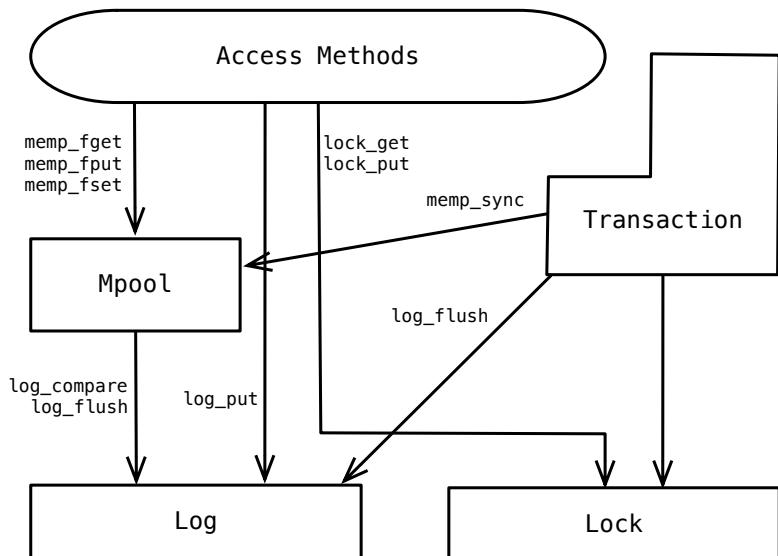



図 4.2: Berkeley DB-2.0 で意図していたアーキテクチャ

LIBTP の実装と Berkeley DB 2.0 の設計との間での唯一の大きな違いは、プロセスマネージャーが削除されたという点である。LIBTP ではコントロールするスレッドが自分自身をライブラリに登録してから個々のスレッド/プロセスを同期させなければならず、サブシステム

レベルでの同期処理は用意していなかった。4.4 節で議論するが、オリジナルの設計のほうがうまく機能したかもしれない。

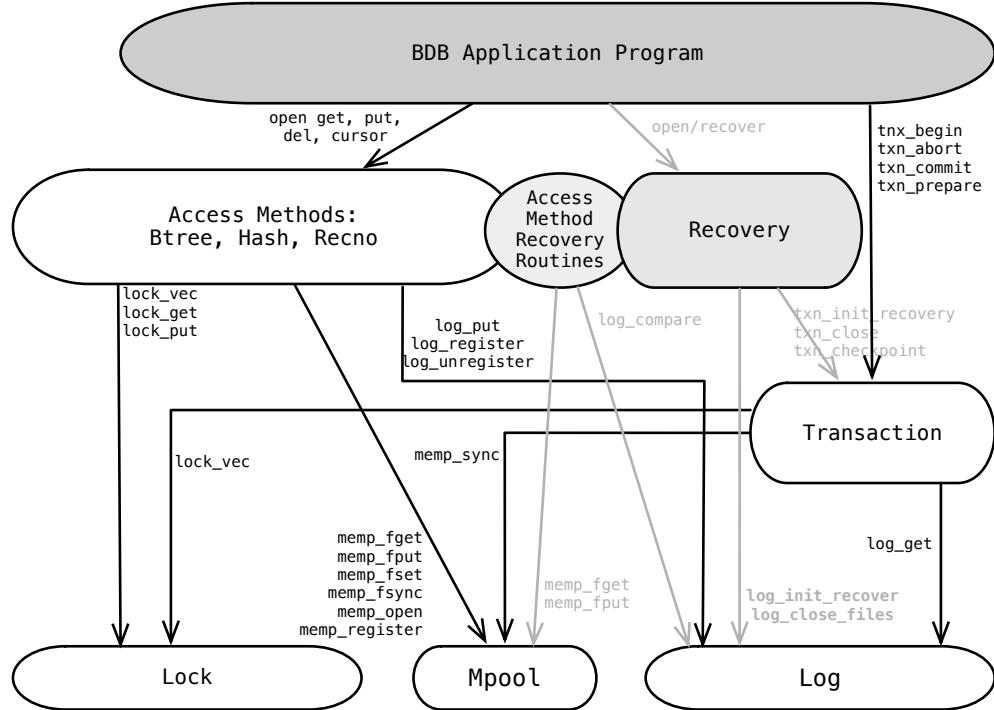



図 4.3: 実際の Berkeley DB 2.0.6 のアーキテクチャ

当初の設計と、実際にリリースされた db-2.0.6 のアーキテクチャ(図 4.3) の違いを見れば、現実には堅牢なリカバリーマネージャーを実装したことがわかる。図の中でグレーで表記されている部分がリカバリーサブシステムである。リカバリー処理には、ドライバの基盤として図中に「recovery」と記されている部分だけではなく redo や undo のルーチンのリカバリーも含まれる。後者は、アクセスメソッドによる操作をリカバーするものだ。後者については "access method recovery routines" と書かれた円で表記している。このように、Berkeley DB 2.0 では一貫性のある設計でリカバリーを扱っている。LIBTP ではこれと対照的に、ログやリカバリーのルーチンを個々のアクセスメソッドごとに手書きしていた。この汎用的な設計により、さまざまなモジュール間でのよりリッチなインターフェイスができあがった。

図 4.4 に Berkeley DB-5.0.21 のアーキテクチャを示す。図中の番号は、表 4.1 にまとめた API の番号を表す。中には当初から変わらない部分も見受けられるが、現在のアーキテクチャは時を経て変化している。新たなモジュールが追加されたり古いモジュールが分割されたり (たとえば、かつての `log` は `log` と `dbreg` に分かれた)、さらにモジュール間の API が激増したりしている。

この 10 年の進化、十回を超える商用リリース、そして何百もの新機能。これらを経て、以

前のバージョンよりもアーキテクチャがかなり複雑化してきた。中でも特筆すべき点をいくつか取り上げる。まずは、レプリケーション機能がまったく新しいレイヤーとしてシステムに追加されたこと。しかしこれは完全にクリーンな状態で追加されており、おもしろいことにシステムの残りの部分で同じ API を使っているところは以前のコードがそのまま動く。二点目が、`log` モジュールを `log` と `dbreg` (database registration) に分割したことだ。この件については 4.8 節で詳しく説明する。三点目が、すべてのモジュール間呼び出しを先頭にアンダースコアをつけた名前空間にまとめたという点だ。これで、システムを使うアプリケーション側が関数名の重複を気にせずに済むようになった。この件については設計講座 6 で解説する。

四点目が、ログ出力サブシステムの API がカーソルベースとなった (`log_get` API がなくなって、`log_cursor` API に置き換えられた) という点だ。Berkeley DB はこれまで、ログの読み書きの制御に複数のスレッドを持つことは一度たりともなかった。したがって、ライブラリはログ内の現在のシーク位置を一か所気にするだけだった。これは決してうまい抽象化であるとは言えず、レプリケーション環境では動作しなくなった。アプリケーションの API がカーソルを使った反復処理に対応したことで、ログもカーソルを使った反復処理をサポートするようになった。五点目が、アクセスメソッドの中の `fileop` モジュールがデータベースの作成・削除・リネームをトランザクション内でできるようにしたことだ。この実装をうまくまとめるために何度か試行錯誤をした (そしてまだ満足のいくレベルにはなっていない) が、何度も作りなおした後でモジュール内に組み込んだ。

設計講座 **2**

ソフトウェアの設計というものは、単に問題の全体像を把握してから解決を試みようという流れを強制するための手段のひとつにすぎない。熟練したプログラマーは、その目的を達成するためにさまざまなテクニックを使う。まずはとりあえず動くものを書いて、最終的にそれを捨ててしまうという人もいるだろう。詳細なマニュアルや設計文書を書くところから始める人もいるだろう。すべての要件を明確化したコードのテンプレートを作ってから、個別に関数やコメントをはめ込んでいく人もいるだろう。Berkeley DB の場合は、まず最初に完全な Unix マニュアルページを作るところから始めた。各アクセスメソッドやその土台となるコンポーネントのマニュアルを、まだ一切コードを書いていない状態で作ったのだ。どんな手法を使うにせよ、コードを書いてデバッグが始まってからプログラムのアーキテクチャについて考えるのは難しい。大規模なアーキテクチャの変更をしようとすると、それまでのデバッグの労力が無駄になってしまうこともよくあるだろう。ソフトウェアのアーキテクチャを考えるときにはコードのデバッグをするときとは異なる考え方が要求される。そして、デバッグを始めた時点でのアーキテクチャが、通常はリリースの時まで引き継がれることになる。

なぜアーキテクトはトランザクションライブラリをコンポーネントから外に出したのだろ

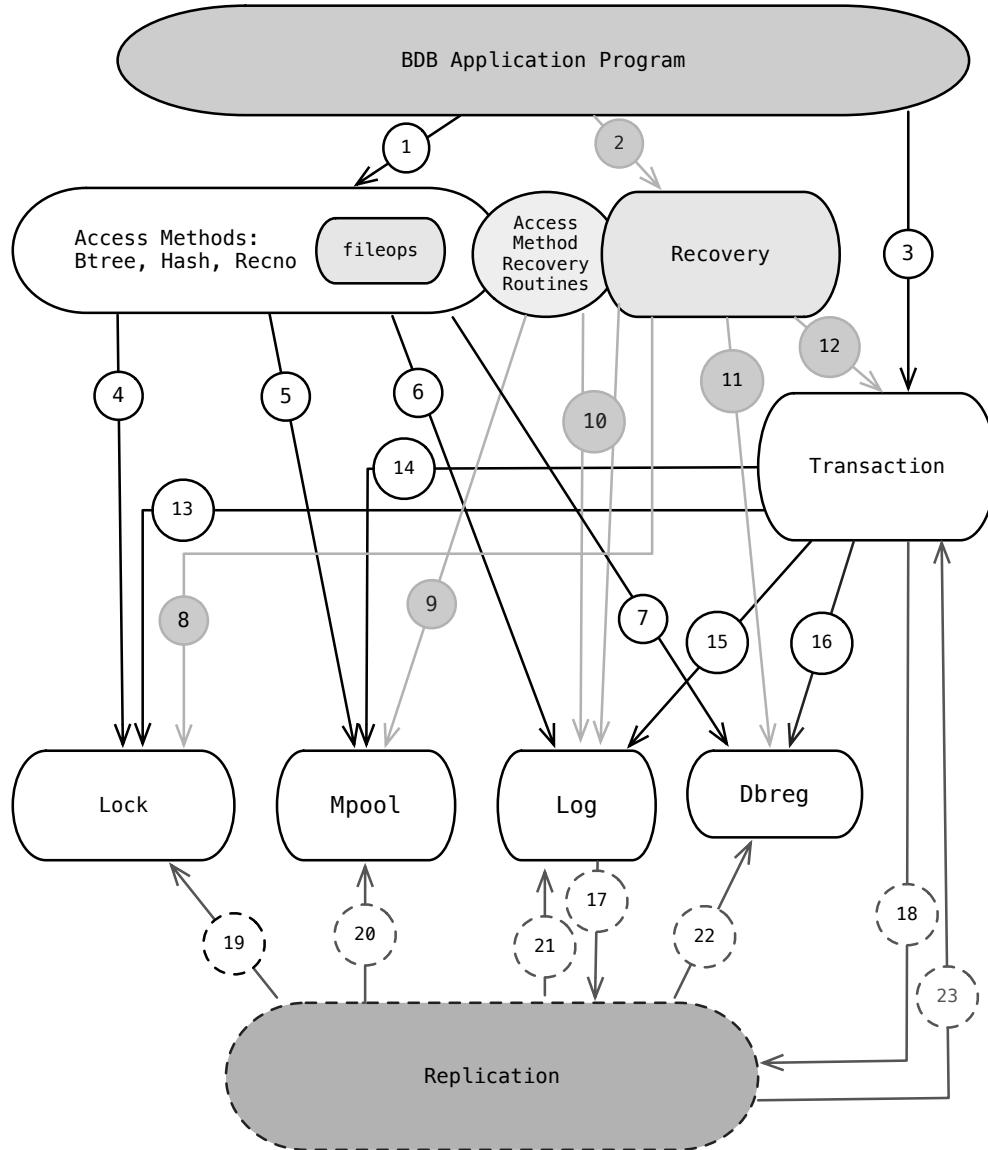



図 4.4: Berkeley DB-5.0.21 のアーキテクチャ

う? 想定されるたったひとつの使い方にあわせてチューニングすればよかったのではないだろうか? この問いへの答えは次の三つである。まず、外に出せばよりしっかりとした設計を強要できる。次に、コードの間にきちんとした境界がなければ、複雑なソフトウェアパッケージは必然的に退化して保守不能な状態になってしまう。最後に、ユーザーがそのソフトウェアをどのように使うかなど、完全に予測するのは不可能だ。ユーザーにソフトウェアコンポーネントへのアクセスを許可すると、きっと思いもよらない方法でそれを使われることになる

アプリケーション API				
1. DBP ハンドル操作 open get put del cursor		2. DB_ENV リカバリー open(... DB_RECOVER ...)		3. トランザクション API DB_ENV->txn_begin DB_TXN->abort DB_TXN->commit DB_TXN->prepare
各アクセスメソッドが使う API				
4. Lock へ __lock_downgrade __lock_vec __lock_get __lock_put	5. Mpool へ __memp_nameop __memp_fget __memp_fput __memp_fset __memp_fsync __memp_fopen __memp_fclose __memp_ftruncate __memp_extend_freelist	6. Log へ __log_print_record	7. Dbreg へ __dbreg_setup __dbreg_net_id __dbreg_revoke __dbreg_teardown __dbreg_close_id __dbreg_log_id	
リカバリー API				
8. Lock へ __lock_getlocker __lock_get_list	9. Mpool へ __memp_fget __memp_fput __memp_fset __memp_nameop	10. Log へ __log_compare __log_open __log_earliest __log_backup __log_cursor __log_vtruncate	11. Dbreg へ __dbreg_close_files __dbreg_mark_restored __dbreg_init_recover	12. Txn へ __txn_getckp __txn_checkpoint __txn_reset __txn_recycle_id __txn_findlastckp __txn_ckp_read
トランザクションモジュールが使う API				
13. Lock へ __lock_vec __lock_downgrade	14. Mpool へ __memp_sync __memp_nameop	15. Log へ __log_cursor __log_current_lsn	16. Dbreg へ __dbreg_invalidate_files __dbreg_close_files __dbreg_log_files	
レプリケーションシステムへの API				
		17. Log から __rep_send_message __rep_bulk_message		18. Txn から __rep_lease_check __rep_txn_applied __rep_send_message
レプリケーション日ステムからの API				
19. Lock へ __lock_vec __lock_get __lock_id	20. Mpool へ __memp_fclose __memp_fget __memp_fput __memp_fsync	21. Log へ __log_get_stable_lsn __log_cursor __log_newfile __log_flush __log_rep_put __log_zero __log_vtruncate	22. Dbreg へ __dbreg_mark_restored __dbreg_invalidate_files __dbreg_close_files	23. Txn へ __txn_recycle_id __txn_begin __txn_recover __txn_getckp __txn_updateckp

表 4.1: Berkeley DB 5.0.21 の API

だろう。

これ以降のセクションでは Berkeley DB の各コンポーネントについて検討し、その動きを理解して全体の中でどのような位置づけになっているのかを把握する。

4.3 アクセスメソッド: Btree, Hash, Recno, Queue

Berkeley DB のアクセスメソッドが提供する機能は、キー指定による検索や反復処理を、可変長バイト文字列および固定長バイト文字列に対して行うことだ。Btree と Hash は、可変長のキー/バリューペアに対応している。Recno と Queue は、レコード番号/バリューペアに対応している (Recno は可変長の値に対応しているが、Queue がサポートするのは固定長の値だ

けである)。

Btree と Hash の主な違いは、Btree がキーの参照の局所性を提供するのに対して Hash はそうではないということだ。つまり、Btree はほぼすべてのデータセットに対して適切なアクセスメソッドとなる。一方、Hash アクセスメソッドが適切に使えるのは、データセットが大きすぎて Btree インデックス構造すらメモリ内に収まらないような場面となる。当時は、メモリはインデックス構造ではなくデータのために使うほうがよかったのだ。このトレードオフは、1990 年の時点では十分納得できるものだった。そのころのマシンは、現在のものと比べてメインメモリの量が圧倒的に少なかったのだ。

Recno と Queue の違いは、Queue がレコードレベルのロックに対応しているという点だ。その代償として、Queue は固定長の値しか受け付けない。Recno は可変長のオブジェクトに対応しているが、Btree や Hash と同様にページレベルのロックしかサポートしていない。

もともとの Berkeley DB の設計は、いわゆる CRUD 機能 (Create:作成、Read:読み込み、Update:更新、Delete:削除) をキーベースで行うもので、それがアプリケーションからの主なインターフェイスだった。そこに後付けでカーソルを追加し、反復処理に対応させた。その結果としてコードが混乱してしまい、ライブラリの内部で同じようなコードの流れが重複してしまうことになった。時を経てそのコードは保守不能になってしまい、キーベースの操作をすべてカーソル操作で書き換えるはめになった (キー指定による操作は、現在はキャッシュカーソルを割り当てて操作を実行してからカーソルをカーソルプールに返すという流れになっている)。これは、ソフトウェア開発の世界で何度となく繰り返されるルールの適用例のひとつである。「コードの可読性を落としたり複雑化させたりしてしまうような最適化は、本当にそれが必要となるまでは決してしてはいけない」ということだ。

設計講座 **3**

ソフトウェアのアーキテクチャが優雅に成熟していくなんてことはない。ソフトウェアのアーキテクチャは、ソフトウェアに加えた変更の数に正比例して退化する。バグ修正のおかげでレイヤー化が崩れたり、新機能の追加で設計に無理が出てきたりといった具合だ。徐々に崩れていくソフトウェアアーキテクチャに対して、どの時点でモジュールの設計を見直して書き直すべきか。これはとても難しい決断だ。アーキテクチャが退化するにつれて、そのソフトウェアの保守や開発はより難しくなる。最終的にはリリースのたびに大量の手動テスト部隊を動員しなければいけないようなレガシーコードの塊になってしまう。ソフトウェアの内部構造を理解できる人が誰もいなくなってしまうのだ。一方、根本的に書き直そうとすると、使う側から見れば不安定で互換性のない状態に悩まされることになる。ソフトウェアアーキテクトであるあなたに対して確実に保障できるのは、どちらの道を選んだところで結局誰かに恨まれるということくらいだ。

Berkeley DB の各アクセスメソッドの内部構造に関する議論はここでは省略する。単に、よ

く知られている Btree やハッシュのアルゴリズムを実装しているにすぎない (Recno は Btree のコードの上位に重ねたレイヤーである。また Queue はファイルブロックのルックアップ処理だが、レコードレベルのロックを追加したために多少複雑になっている)。

4.4 ライブラリインターフェイス層

長年にわたって機能追加を続けていくうちに、ようやく気付いたことがある。アプリケーションのコードと内部のコードとで同じ機能を共有する必要があるということだ (たとえばテーブルの JOIN 操作では複数のカーソルを使って行の反復処理を行う。一方アプリケーション側でも、カーソルを使って同じ行を反復処理することになるだろう)。

設計講座 **4**

変数やメソッドや関数の名前の付けかた、コメントの書きかた、そしてコードの書きかた。どんなスタイルを選んでもかまわない。世の中には「よい書きかた」とされている書式やスタイルがいろいろある。そんなことより大切なのは、どう書くかではなく命名規約やコーディングスタイルの一貫性を保つことだ。熟練したプログラマーは、コードのフォーマットやオブジェクトの命名から大量の情報を引き出す。命名規約やスタイルが一貫していないと、コードの内容を読み取るのに時間と労力がかかってしまい、他のプログラマーに誤読されてしまうことになるだろう。内部的なコーディング規約に反するのは、そのシステムに対して発砲しているのに等しい。

というわけで、アクセスメソッド API を分割してきちんと定義された階層に分けることにした。インターフェイスルーチン層は、必要となる汎用的なエラーチェックや関数固有のエラーチェック、インターフェイスの追跡などをすべて行う。それ以外にも、自動トランザクション管理などもこの層の仕事だ。アプリケーションが Berkeley DB の機能を利用するときには、処理対象のオブジェクトが持つメソッドにもとづいた第一レベルのインターフェイスを呼び出す。(たとえば、`__dbc_put_pp` は Berkeley DB のカーソルの “put” メソッドに対応するインターフェイスの呼び出しで、データアイテムを更新する。最後の “_pp” は、アプリケーションから呼び出せる関数であることを示すサフィックスである)。

Berkeley DB がインターフェイス層で行うタスクの一つが、どのスレッドが Berkeley DB ライブラリ内で動作中なのかを追跡することだ。追跡が必要となる理由は、Berkeley DB の内部的な操作の中にはライブラリ内でスレッドが立ち上がっていないときにしか実行できないものもあるからである。Berkeley DB がライブラリ内でのスレッドの動きを追跡する方法は、ライブラリの API をコールするたびにそのスレッドが実行中であるというフラグを立てておき、API コールの結果が返ってきた時点でフラグをクリアするというものだ。この「入場/退

場のチェック」が、常にインターフェイス層で行われる。これは、そのコールがレプリケーション環境から行われたのかをチェックするのと同じ方法である。

当然、こんな疑問が出てくるはずだ。「スレッド ID を直接ライブラリに渡したほうがずっと簡単なのでは?」答えはイエスだ。そのほうがずっと簡単だろうし、できることならそうしたかった。しかし、そんな変更をしてしまうと、Berkeley DB を使ったアプリケーションはすべて修正が必要となる。アプリケーションからの Berkeley DB の機能の呼び出しの大半を書き換えることになり、たいていの場合はアプリケーションの構造を再考することになるだろう。

設計講座 5

ソフトウェアアーキテクトは、アップグレードに伴ういざこざへの対応に気をつける必要がある。新しいリリースに対応するために要する変更が少しで済むのなら、ユーザーはそれを許容するだろう (コンパイル時にエラーが出るようにしておけば、まだアップグレードが完了していないことが明白になる。アップグレードの際によくわからないエラーで悩まされることもなくなるだろう)。しかし、まったく根本的に変化させるには、新しいコードベースを使わざるを得ない。そしてユーザーにも新たなコードベースへの移行を求めることになる。明らかに、アプリケーションを一から書き直してユーザーの移行を促すのは、時間的にもリソース的にもたやすいことではない。しかし、実際にはちょっとしたアップグレードなのに大規模な改修が必要となってユーザーを怒らせるといったことはしなくて済む。

インターフェイス層で行われるもうひとつのタスクが、トランザクションの生成だ。Berkeley DB ライブラリがサポートするモードのひとつに、すべての操作が自動生成されたトランザクション内で行われるというモードがある (これによって、アプリケーション側で明示的にトランザクションを作成したりコミットしたりする手間を省ける)。このモードをサポートするには、アプリケーション側からトランザクションを明示せずに API をコールされるたびに、トランザクションを自動生成することになる。

最後に、すべての Berkeley DB API は引数のチェックを要する。Berkeley DB のエラーチェックには二種類ある。ひとつは汎用的なチェックで、直近の操作でデータベースが破壊されていないかどうかを調べたりレプリケーションの状態が変わっている最中 (たとえば、書き込み可能なレプリカの切り替え中など) であるかどうかを調べたりといったものだ。もうひとつは個々の API に固有のチェックで、フラグの用法やパラメータの使い方、オプションの組み合わせ、その他実際にリクエストを処理する前に確認できるあらゆるエラーのチェックを行う。

API 固有のチェックはすべて、最後に`_arg`がつく名前の関数にカプセル化されている。つまり、カーソルの`put`メソッドに関するエラーチェックは`__dbc_put_arg`関数にまとめられており、これが`__dbc_put_pp`関数から呼び出される。

最終的に、引数の検証やトランザクションの生成が完了したら、実際の処理を行うワーカーメソッド (今回の場合は__dbc_put) を呼び出す。これは、我々がカーソルの put 機能を内部的に使うときに利用するのと同じ関数である。

このように分割することで、大がかりな操作があったときにレプリケーション環境で実際にどんなアクションが必要となるのかを判断しやすくなった。コードベースに手を加える作業を数え切れないほど繰り返した結果、すべての事前チェックを分離することができた。そして、今後もし何か問題が発生したときにもより手を加えやすくなったのだ。

4.5 基盤となるコンポーネント

アクセスメソッドの基盤となるのが、バッファマネージャー・ロックマネージャー・ログマネージャーそしてトランザクションマネージャーの四つのコンポーネントだ。それぞれについて個別に扱うが、これらすべてに共通するアーキテクチャ上の特性もある。

まず、すべてのサブシステムには自前の API が存在する。当初は各サブシステムが自身のオブジェクトハンドルを保持しており、そのハンドルでサブシステムのすべてのメソッドを扱っていた。たとえば Berkeley DB のロックマネージャーを使うと、自身のロックを処理したりリモートのロックマネージャーに書き出したりできる。あるいは Berkeley DB のバッファマネージャーを使うと、自身のファイルページを共有メモリで扱える。時を経て、これらのサブシステム固有のハンドルは API から削除された。Berkeley DB を使うアプリケーションをシンプルにするためだ。サブシステム群は今でも個別のコンポーネントだし他のサブシステムとは独立して使えるようになっているが、今では共通のオブジェクトハンドルである DB_ENV "environment" ハンドルを共有するようになった。このアーキテクチャが、レイヤー化と汎化を強制する。レイヤーの区切りは時とともに変化するし、あるサブシステムが別のサブシステムの役割に立ち入っている部分もいくつか存在する。しかし、プログラマーにとっては、システムの各部分を個別のソフトウェアプロダクトとして考えるのはよいことだ。

次に、すべてのサブシステム (実際のところ、すべての Berkeley DB の関数) はエラーコードをコールスタックに返す。Berkeley DB はライブラリであり、グローバル変数を宣言したりしてアプリケーションの名前空間に立ち入ることができない。言うまでもないが、すべてのエラーをコールスタック経由の単一のパスで返すことはよい習慣である。

設計講座 6

ライブラリを設計する際には、名前空間を利用することが大切である。そうしなければ、ライブラリを使う側のプログラマーからすれば、関数や定数そして構造体やグローバル変数の名前を何十個も覚えておかないとアプリケーション側で名前が衝突してしまうことになる。

最後に、すべてのサブシステムは共有メモリをサポートする。Berkeley DB は複数のプロ

セスからのデータベースの共有をサポートしているので、すべてのデータ構造は共有メモリ上に存在することになる。この方式を選んだことによる最大の影響は、インメモリのデータ構造がポインタではなくベースアドレスとオフセットのペアを使わなければならないということだ。これは、ポインタベースのデータ構造をマルチプロセスのコンテキストでも動作させるために必要となる。言い換えると、ポインタ経由で間接参照するのではなく、Berkeley DB ライブラリはベースアドレス (メモリ内で共有メモリセグメントがマップされたアドレス) にオフセット (マップされたセグメントの中で、そのデータ構造が存在する位置) を足した場所を参照しなければならないということである。この機能をサポートするために、我々は Berkeley Software Distribution queue パッケージの別バージョンを書いた。これはさまざまな種類のリンクリストを実装したものである。

設計講座 **7**

実際に共有メモリのリンクリストパッケージを書き始める前に、Berkeley DB エンジニアは共有メモリ内でのさまざまなデータ構造を手書きしていた。しかしこれらの実装はどれも脆く、デバッグしにくいものだった。共有メモリのリストパッケージは BSD のリストパッケージ (`queue.h`) を参考にして作ったもので、今まで苦労した結果をこれがすべて置き換えてくれた。一度デバッグをしてからは、共有メモリのリンクリストに関する問題は二度と発生しなかった。このことから得られる重要な設計指針は次の三つだ。まず、もし複数回登場する機能があるのなら、共有関数を作ってそれを使うようにする。なぜなら、コードの中に同じような機能が複数回登場するということはそのうちのどれかが間違った実装になっているだろうからだ。次に、汎用目的のルーチンを開発するときにはそのルーチン群用のテストスイートを書く。そうすれば、そのルーチンだけを個別にデバッグできる。最後に、書くのが難しいコードであればあるほど、その部分を個別に書いて保守していくことが重要になる。そうしないと、その周辺のコードの影響を受けて浸食されていることを防げなくなる。

4.6　バッファマネージャー: Mpool

Berkeley DB の Mpool サブシステムは、ファイルページを扱うインメモリのバッファプールである。メインメモリが有限なリソースであるという事実を隠し、メモリの量を超える大きさのデータベースを扱うときにはデータベースのページをディスクとの間で移動させる。データベースのページをメモリ内でキャッシュすることで、オリジナルの hash ライブラリの性能は劇的に向上した。かつての `hsearch` や `ndbm` の実装の性能を大きく上回る。

Berkeley DB の Btree アクセスメソッドは極めて伝統的な B+木の実装であるが、ツリーノード間のポインタはページ番号で表しており、実際のインメモリのポインタではない。その理

由は、ライブラリの実装がインメモリだけではなくディスク上でのフォーマットも扱うからである。このようにする利点は、ページをキャッシュからフラッシュするときにフォーマット変換が不要になることだ。逆に欠点は、インデックスを走査する際にバッファプールのルックアップが必要になるということだ。バッファプールのルックアップは、間接メモリ参照よりもコストの高い操作である。

それ以外にも、パフォーマンスに関する暗黙の影響がある。それは、インメモリでのBerkeley DBのインデックスの表現方法が実際のところはディスク上の永続データのキャッシュであるという前提から得られるものだ。たとえば、Berkeley DBがキャッシュされたページにアクセスするときには、まずメモリ内のページを固定する。固定することで、他のスレッドやプロセスがそのページをバッファプールから削除できないようにする。インデックスが完全にキャッシュ内に収まってディスクへのフラッシュが不要だったとしても、Berkeley DBはアクセスのたびにページを固定する。その理由は、Mpoolが提供する基盤モデルがキャッシュであって永続ストレージではないからだ。

Mpoolのファイル抽象化

Mpoolはファイルシステム上で動かすことを前提としており、ファイルを抽象化したAPIを公開している。たとえば`DB_MPOOLFILE`ハンドルはディスク上のファイルを表し、ファイルからページを取得したりページをファイルに書き出したりといったメソッドを提供する。Berkeley DBではインメモリのテンポラリデータベースに対応しているが、これらもまた`DB_MPOOLFILE`ハンドルから参照されている。これは、ベースになるMpoolの抽象化によるものだ。Mpoolの主要なAPIは`get`メソッドと`put`メソッドである。`get`はページがキャッシュ内にあることを確かめてからそのページを固定し、そしてページへのポインタを返す。ライブラリがそのページでの処理を終えたら、`put`を呼んでページの固定を解除し、明け渡す。初期のBerkeley DBは、読み込み用にページを固定するときと書き込み用にページを固定するときを区別していなかった。しかし、同時実行性を向上させるためにMpoolのAPIを拡張し、呼び出し側でページの書き込み意図を明示できるようにした。このように読み込みアクセスと書き込みアクセスを区別できることは、多版型同時実行制御を実装するために不可欠だ。読み込み用として固定したページはダーティになったとしてもディスクに書き込めるが、書き込み用に固定した場合はできない。整合性を欠く状態になっている可能性があるからである。

ログ先行書き込み

Berkeley DBは、ログ先行書き込み（WAL）をトランザクションの仕組みとして使っており、処理に失敗した場合の復旧をできるようにしている。ログ先行書き込みという用語は、実際にデータを更新する**前に**変更内容のログをディスクに書き込ませるという方針を表している。Berkeley DBがWALをトランザクション機構に採用しているということはMpoolに

とって大切な前提となる。Mpool は、汎用的なキャッシュの機能と WAL プロトコルのサポートとの間でバランスをとった設計でなければならない。

Berkeley DB は、すべてのデータページにログシーケンス番号（LSN）を書き込む。これを使って、特定のページへの直近の変更に対応するログレコードを記録するのだ。WAL を強制すると、Mpool がページをディスクに書き出す前にそのページの LSN に対応するログレコードがディスク上にあることを確認しなければならない。ここで設計上の難題が発生する。この機能を提供しつつ、Mpool のクライアントで Berkeley DB とは違うページフォーマットを使えるようにするにはどうすればいいだろうか。Mpool がこの難題にこたえるために用意したメソッドが `set`（および `get`）のコレクションだ。これを使って振る舞いを指示する。`DB_MPOOLFILE` のメソッド `set_lsn_offset` はページ内のバイトオフセットを提供し、どこに LSN があるのかを Mpool に指示する。これは、WAL を確実にするために必要なものだ。このメソッドが一切呼ばれなければ、Mpool は WAL プロトコルを強要しない。同様に `set_clearlen` メソッドは、ページをキャッシュ内で作るときにメタデータを何バイトぶん明示的にクリアする必要があるのかを Mpool に伝える。これらの API のおかげで、Mpool は Berkeley DB のトランザクション要件に対応した機能を提供できるようになる。Mpool を使う全ユーザーに対応を要求せずに済むのだ。

設計講座 **8**

ログ先行書き込みはカプセル化とレイヤー化の一例でもある。たとえその機能がソフトウェアの他の箇所では一切使わないものだったとしても。結局のところ、キャッシュの中の LSN を気にかけているプログラムがいったいどれほどあるのだというのだろう？にもかかわらず、こういう指針は有用だ。ソフトウェアの保守やテスト、デバッグ、拡張をしやすくしてくれる。

4.7　ロックマネージャー: Lock

Mpool と同様、ロックマネージャーも汎用コンポーネントとして作られた。階層化ロックマネージャー（[GLPT76] を参照）としてオブジェクトの階層（個々のデータ項目など）のロックに対応しており、それだけでなくデータ項目が存在するページやファイルあるいはファイルのコレクションにも対応している。ロックマネージャーの機能を説明するとともに、Berkeley DB がそれをどう活用しているのかについても解説する。しかし、Mpool の場合と同様、他のアプリケーションからもまったく違う方法でロックマネージャーを使えるという点が重要だ。もともとそういった使い方を想定して柔軟に作られており、さまざまな使い方に対応している。

ロックマネージャーが抽象化している重要な概念は次の三つ。“ロッカー（locker）” は誰がロックを獲得したのかを表し、“ロックオブジェクト（lock_object）” はロックされれてい

る項目を表す。そして最後が “衝突マトリクス（conflict matrix）” である。

ロッカーは 32 ビット符号なし整数である。Berkeley DB は、この 32 ビットの名前空間トランザクショナルなロッカーと非トランザクショナルなロッカーの二つに分割する（ただしロックマネージャー側からはこれらを区別しない)。Berkeley DB がロックマネージャーを使うときは、非トランザクショナルなロッカーについては 0 から 0x7fffffff、トランザクショナルなロッカーについては 0x80000000 から 0xffffffff の範囲のロッカー ID をトランザクションに割り当てる。たとえば、あるアプリケーションがデータベースを開いたとする。このとき Berkeley DB はそのデータベースに対して長期間の読み込みロックを確保し、利用中は他の制御スレッドからの削除やリネームができないようにする。このロックは長期間のロックなのでトランザクションには属さず、ロッカーはこのロックを非トランザクショナルとして保持する。

ロックマネージャーを使うすべてのアプリケーションは、ロッカー ID を割り当てる必要がある。そこで、ロックマネージャーの API では `DB_ENV->lock_id` と `DB_ENV->lock_id_free` の両方を提供している。これらを呼べば、ロッカーの割り当てと解放ができる。なので、アプリケーション側ではロッカー ID の割り当て処理の実装は不要だ。

ロックオブジェクト

ロックオブジェクトは任意の長さの不透明なバイト文字列で、これがロック対象のオブジェクトを表す。二つの異なるロッカーが特定のオブジェクトをロックしようとしたときは、どちらも同じバイト文字列でそのオブジェクトを表す。つまり、オブジェクトを不透明なバイト文字列で表すときの規約についてはアプリケーション側で決めておく必要がある。

たとえば Berkeley DB では、DB_LOCK_ILOCK 構造体をつかってデータベースロックを表す。この構造体に含まれるフィールドは三つ。ファイル ID とページ番号、そして型である。

ほとんどの場合、Berkeley DB で表す必要があるのはロックしたい特定のファイルやページだけである。Berkeley DB はデータベースの作成時に一意な 32 ビット数値を割り当てており、データベースのメタデータページにそれを書き込んでいる。Mpool やロック、ログなどのサブシステムでは、これをデータベースの一意な ID として使う。DB_LOCK_ILOCK 構造体の `fileid` が、その ID である。おそらく予想通りだろうが、ページ番号が指すのはデータベース内でロックしたいページである。ページロックを指すときには、構造体の type フィールドに DB_PAGE_LOCK を設定する。しかし、必要に応じて他の型のオブジェクトもロックできる。先述のとおり、時にはデータベースのハンドルをロックすることもあり、その場合に指定する型は DB_HANDLE_LOCK となる。DB_RECORD_LOCK 型を使えば queue アクセスメソッドでレコードレベルのロックができ、DB_DATABASE_LOCK 型を使えばデータベース全体をロックする。

> **設計講座 9**
>
> Berkeley DB がページレベルのロックを選んだことにはそれなりの理由がある。しかし、時にはその選択が問題となることも出てきた。ページレベルのロックはアプリケーションの同時実行性を制限してしまう。ある制御スレッドがデータベースのレコードを変更すると、他のスレッドからは同じページにあるそれ以外のレコードを変更できなくなる。一方レコードレベルのロックなら、全く同じレコードを変更しようとしない限りはそんな制限を受けない。ページレベルのロックをすれば安定性が向上する。考えうるリカバリーパスが絞り込めるからである（リカバリ中は、ページは常にいくつかの状態の中のひとつでいる。しかし、もしひとつのページの中で複数のレコードが追加されたり削除されたりしていたら、無数の可能性が存在しうることになる）。Berkeley DB は組み込みシステムでの利用を想定して作られたものである。そんな環境では、何か問題があったときに対応してくれるようなデータベース管理者がいるとは限らない。そこで私たちは、同時実行性よりも安定性を重視することにしたのだ。

衝突マトリクス

ロックのサブシステムの中で最後に取り上げる抽象概念が、衝突マトリクスだ。これは、システム内に存在するさまざまな型のロックとその相互作用について定義する。ロックを保持するエンティティを「ホルダー」、ロックを要求するエンティティを「リクエスター」と呼び、ホルダーとリクエスターがそれぞれ別のロッカー ID を持っているものとする。このとき、衝突マトリクスは `[requester][holder]` をインデックスとする配列になる。もし衝突がなければそのエントリの値は 0 で、これはロックが認められることを表す。衝突が発生する場合の値は 1 で、これは要求が認められなかったことを表す。

ロックマネージャーにはデフォルトの衝突マトリクスが組み込まれており、これはたまたま Berkeley DB が必要とするものと完全に一致している。しかし、アプリケーション側で目的に合わせて自前のロックモードと衝突マトリクスを作ってもかまわない。衝突マトリクスに関する条件は、正方行列である（行の数と列の数が等しい）ことと、ロックモード（read、write など）をゼロから始まる整数の連番で表すということだけである。表 4.2 に、Berkeley DB の衝突マトリクスを示す。

階層化ロックのサポート

Berkeley DB の衝突マトリクスにおけるさまざまなロックモードについて説明する前に、ロックサブシステムがどうやって階層化ロックに対応しているのかを説明しよう。階層化ロックとは、さまざまな項目を階層を保ったままロックすることだ。たとえば、ファイルの中には複数のページがあり、ページの中には個々の要素が含まれる。あるページの要素を階層化

リクエスター	ホルダー **No-Lock**	**Read**	**Write**	**Wait**	**iWrite**	**iRead**	**iRW**	**uRead**	**wasWrite**
No-Lock									
Read			✓		✓		✓		✓
Write		✓	✓	✓	✓	✓	✓	✓	✓
Wait									
iWrite		✓	✓					✓	✓
iRead			✓						✓
iRW		✓	✓					✓	✓
uRead			✓		✓		✓		
iwasWrite		✓	✓		✓	✓	✓		✓

表 4.2: Read-Writer 衝突マトリクス

ロックのもとで変更するときは、その要素だけをロックしておきたい。ページ上の全要素を変更するなら、各要素をロックするよりも単にそのページを丸ごとロックしたほうが効率的だ。同様に、ファイル内のすべてのページを変更するのならファイル全体をロックするのがいちばんだ。さらに、階層化ロックを行うにはコンテナの階層についても知っておく必要がある。あるページをロックするということは、ファイルをロックするということでもあるからだ。あるページが変更されているとき、そのページを含むファイルを同時に変更することはできない。

というわけで、問題は、さまざまなロッカーが異なる階層レベルを見るときに混乱を引き起こさずに済ませるにはどうすればいいか？ということになる。その答えはインテンションロックと呼ばれる概念にある。あるロッカーがコンテナに対するインテンションロックを獲得すると、そのコンテナの中身に対してもロックする意図があるということを表す。つまり、ページへの書き込みロックを獲得するということは、ファイルへの書き込みインテンションロックを獲得することを意味する。同様に、あるページのひとつの要素に書き込むには、ページとファイルの両方に対する書き込みインテンションロックを獲得しなければならない。さきほどの衝突マトリクスにおいて `iRead`、`iWrite`、そして `iWR` はすべてインテンションロックであり、それぞれ読み込み、書き込み、その両方に対する意図を指す。

したがって、単に何かを単体でロックするのではなく階層化ロックを実行しようとした場合は、多数のロックを要求しなければならなくなる。実際にロックしたいエンティティだけでなく、それを含むエンティティに対するインテンションロックも必要となる。そのためには Berkeley DB の `DB_ENV->lock_vec` インターフェイスが必要だ。これは、ロック要求の配列を受け取って、その承認（あるいは却下）をアトミックに行う。

Berkeley DB は内部的には階層化ロックを使っていないが、異なる衝突マトリクスを指定できるという機能や複数のロック要求を一度に指定できる機能は活用している。デフォルトの衝突マトリクスはトランザクションをサポートするときに使うが、トランザクションやリカバリーのサポートをやめてシンプルな同時アクセス機能を提供するときには別の衝突マトリクスを使う。我々は `DB_ENV->lock_vec` を使ってロックの結合をする。これは Btree の走査の同時並行性をあげるためのテクニックだ [Com79]。ロックを結合するときには、ひとつの

ロックを保持するのは次のロックを獲得するまでの間となる。つまり、内部の Btree ページをロックするのは、次のレベルのページを選んでロックするための情報を読み込むまでの間である。

設計講座 **10**

Berkeley DB を汎用性重視の設計にしておいたことが、並列データ格納機能を追加するときに役立った。当初 Berkeley DB が用意していた操作モードは二種類だけで、並列書き込みを一切考慮しないモードと完全なトランザクションに対応したモードだけだった。トランザクションに対応すると開発者にとってはそれなりに複雑になってしまうし、アプリケーションによっては、トランザクションのサポートが不完全でもいいから並列処理を改善してほしいという場合もあった。この機能を提供するために、まずは API レベルのロックに対応して並列処理をできるようにし、デッドロックが発生しないことを保証した。そのために必要なのは、新たなロックモードを別途用意してカーソル上で動作させるということだった。これを実現するために新たなコードをロックマネージャーに追加する必要はなく、API レベルのロックに必要なロックモードだけに対応したロックマトリクスを用意するだけでよかった。このようにして、単にロックマネージャーの設定を調整するだけで、必要となるロックサポートを提供できたのだ（残念ながら、アクセスメソッドの変更は、それほどお手軽にはできなかった。アクセスメソッド側には、並列アクセスのための特別なモードに対応するためのコードが大量に存在する）。

4.8　ログマネージャー: Log

ログマネージャーは、構造化された追記限定のファイルを抽象化したものである。他のモジュールと同様、汎用的なログ機能を念頭に置いて作っている。しかしログ出力サブシステムは、各種モジュール群の中でおそらく最もできの悪いものであろう。

ログの概念は極めてシンプルである。不透明なバイト文字列を受け取り、それをシーケンシャルにファイルへ書き出し、各レコードに一意な識別子を付与する。この識別子はログシーケンス番号 (LSN) と呼ばれる。さらに、この LSN を使って両方向への効率的な走査やレコードの取得をできるようにしておく必要がある。注意すべき点は次の二カ所だ。まず、考え得るあらゆる障害があった後でもその一貫性を保証しなければならない (ここでいう一貫性とは、障害の前後の一連のレコードは正常な状態で残らなければいけないということである)。次に、ログレコードはトランザクションのコミットのたびに安定したストレージに書き出す必要があるので、ログのパフォーマンスはトランザクションを利用するあらゆるアプリケーションに影響する。

> **設計講座 11**
>
> アーキテクチャ上の問題が見つかったけれども「とりあえず今のところは」修正したくない。できればそのままにしておきたいなあ…だって？アヒルのくちばしでの突っつきも、数え切れないほど続いたら死に至るかもよ。そう、ゾウに踏みつぶされたら死んでしまうのと同じように。ソフトウェアの構造を改良するため根本的にフレームワークを変更してしまうことを恐れてはいけない。また、一部だけ変更して後で徐々に整えていこうなんて考えではいけない。やるなら全体的にやってしまって、それから次に進もう。何度となく言われているように、「今それをする時間がないのなら、いつまでたってもそんな時間はひねり出せないだろう」ってことだ。また、フレームワークを変更するときにはテストもきちんと書いておこう。

ログは追記しかしないデータ構造なので、何の束縛もなしにサイズを増やせる。我々はログを連番のファイルとして実装したので、ログ用のスペースを確保するには単に古いログファイルを削除するだけでよい。ログのアーキテクチャを複数ファイル形式にしたので、LSN の形式はファイル番号とファイル内オフセットのペアとした。したがって、LSN さえわかれば、ログマネージャーは簡単にレコードを見つけられる。指定されたファイルの指定されたオフセットまでシークし、そこに書かれているレコードを返すだけのことだ。しかし、場所はわかったとして、そこから何バイトぶん返せばいいのかはどうやって知るのだろう?

ログレコードの書式

ログには必ずレコード単位のメタデータが含まなければならない。そうすれば、LSN さえわかれば返すレコードのサイズもわかるようになる。最低限、レコードの長さが取得できなければいけない。我々が選んだのは、すべてのログレコードの前にレコードヘッダを付加する方法だ。レコード長や前のレコードからのオフセット (逆方向の走査用)、そしてログレコードのチェックサム (ログの破壊やファイルの終端を検出するため) といった情報がヘッダに含まれる。このメタデータがあれば、ログマネージャーがレコードの並びを保守するには十分だ。しかしこれだけでは、リカバリー処理を実装することができない。リカバリー機能はログレコードの中身に組み込まれており、Berkeley DB はログレコード自体を使ってリカバリー処理を行う。

Berkeley DB は、ログマネージャーを利用して更新前・更新後のイメージをそれぞれ記録してからデータベース内のアイテムを更新する [HR83]。このログレコードには、データベース上での操作を取り消したりやり直したりするために必要な情報が含まれている。Berkeley DB はこのログを利用して、トランザクションのアボート (それまで行われた変更をすべて取り消してトランザクションを破棄する処理) やアプリケーションあるいはシステムが異常終

了したときのリカバリーを行う。

ログレコードの読み書き用の API 以外に、ログレコードを強制的にディスクに書き込む API (`DB_ENV->log_flush`) も用意されている。この API を利用して、Berkeley DB のログ先行書き込みを実装しているのだ。ページを Mpool から削除する前に、Berkeley DB がページの LSN を調べ、その LSN がストレージ上に格納されているかどうかをログマネージャに問い合わせる。それが確認できたときにだけ Mpool がページをディスクに書き出すのだ。

設計講座 **12**

Mpool と Log は内部処理のメソッドでログ先行書き込みを利用しており、場合によってはメソッドの宣言のほうが実際のコードよりも長くなることもある。メソッドで実際に行っているのが単なる積分値の比較だけだということが多いからである。一貫したレイヤー構造を保つためだけに、わざわざそこまでする必要があるのかって? 歯が痛くなるほどのオブジェクト指向じゃなければ、十分にオブジェクト指向しているとは言えないのだよ。あらゆるコード片は、あまり多くのことをやりすぎないようにしなければならない。また、小規模な機能のかたまりを取りまとめて一つの機能を構築させるよう、上位レベルの設計をしていかなければいけない。過去数十年間でソフトウェア開発について学んだことがあるとすれば、大量のソフトウェア部品を組み上げたり保守したりする能力はとてもはかないものだということである。大量のソフトウェア部品を組み上げたり保守したりするのは難しくて間違いを犯しやすい。そして、ソフトウェアアーキテクトとして、できる限りのことをしてできるだけ早くできるだけ頻繁にソフトウェアが運ぶ情報を最大化しなければならない。

Berkeley DB は、リカバリー機能を提供できるようなログレコードの構造を強いている。Berkeley DB のログレコードの大半は、トランザクションの更新を記録するものである。したがって、ログレコードの大半はデータベースのページを書き換えるものであり、トランザクション上で実行されていることになる。これを踏まえると、Berkeley DB のログレコードに付加すべきメタデータが決まる。それがデータベース・トランザクション・レコードタイプである。トランザクション ID およびレコードタイプのフィールドは、すべてのレコードで同じ位置に存在する。こうすることで、リカバリーシステムがレコードタイプを取得し、レコードの処理を適切なハンドラに振り分けて正しく処理できるようになる。トランザクション ID を見ればそのログレコードがどのトランザクションに属するのかがわかるので、リカバリー中にそのレコードを無視できるのか処理が必要なのかも判断できる。

抽象化違反

「特別な」ログレコードがいくつかある。その中でもおそらく最もなじみ深いのが、チェックポイントレコードだろう。チェックポイントとは、ある特定時点のデータベースの状態をディスク上に書き出す処理のことである。言い換えれば、Berkeley DB はかなりアグレッシブにデータベースのページを Mpool にキャッシュしているということだ。これはパフォーマンスを稼ぐためである。しかし、キャッシュしたページもいずれはディスクに書き出さねばならず、早めに書けば書くほど障害時のリカバリーを素早く行えるようになる。これらを考慮すると、チェックポイントの頻度とリカバリーに要する時間との間にトレードオフが発生することがわかる。つまり、チェックポイントを頻繁に行えば行うほどリカバリーが素早くできるようになる。チェックポイントはトランザクションの機能なので、その詳細は次のセクションで扱う。このセクションは、まずチェックポイントレコードについて解説する。そして、スタンドアロンのモジュールの場合と特化型の Berkeley DB コンポーネントの場合にログマネージャーがそれをどう扱うのかを見ていく。

一般に、ログマネージャー自身はレコードタイプを気にしないものだ。したがって、チェックポイントレコードだろうがそれ以外のレコードだろうが区別してはいけない。どちらも単なる不透明なバイト文字列であり、ログマネージャーは単にそれをディスクに書き込むだけのことである。実際には、ログはいくつかのレコードの内容を理解するようなメタデータを維持する。たとえばログを立ち上げるときに、ログマネージャーがすべてのログファイルを吟味して直近に書き込まれたログファイルがどれであるかを調べる。それより前のログファイルはすべて完全で壊れていないものとみなし、さらに最新のログファイル内に有効なログレコードが何件あるかを調べる。ログファイルの先頭から読み込み、チェックサムが適切に設定されていないレコードヘッダが見つかった時点で停止する。つまりそこが、ログの終端あるいはログファイルが壊れている場所ということだ。いずれにせよ、ログの論理的な終端はその位置になる。

ログを読んで現在の終端位置を探す処理の間に、ログマネージャーは Berkeley DB のレコードタイプを展開してチェックポイントレコードを探す。ログマネージャーのメタデータの中で見つけた最後のチェックポイントレコードの位置を、トランザクションシステム用に保持する。どういうことかというと、まずトランザクションシステムは直近のチェックポイントの位置を知る必要がある。しかし、ログマネージャーとトランザクションマネージャーがそれぞれログをまるごと読み込んで位置を調べるのではなく、トランザクションマネージャーがその処理をログマネージャーに委譲するということだ。抽象化による処理の切り分けに違反してでもパフォーマンスを求めたという、ありがちな話だ。

このトレードオフから、いったい何が導けるだろう? たとえば、Berkeley DB 以外のシステムがこのログマネージャーを使う場面を想像してみよう。Berkeley DB がレコードタイプを書き込んでいるのと同じ位置に、たまたまチェックポイントレコードタイプに対応する値が書かれてしまったとすると、ログマネージャーはそのレコードをチェックポイントだと解釈してしまう。しかし、アプリケーション側から (ログのメタデータの `cached_ckp_lsn` フィー

ルドに直接アクセスして) 問い合わせない限り、この情報は何も影響を及ぼさない。要するにこれは、有害なレイヤー化違反あるいは抜け目のないパフォーマンス最適化だということだ。

ファイル管理もまた、ログマネージャーと Berkeley DB の役割分担が曖昧になっているところのひとつだ。先述のとおり、Berkeley DB のログレコードの大半はデータベースを特定できなければならない。各ログレコードにデータベースファイルのフルパスを含めてもよいが、そんなことをすればログの容量が肥大化するしあまり美しくない。リカバリーのときにはそのファイル名をデータベースアクセス用のハンドル的な何か (ファイルディスクリプタあるいはデータベースハンドル) に変換する処理が必要になってしまうからだ。そのかわりに Berkeley DB が使っているのが、ログに一意な整数値を記録する方法だ。これはファイル ID と呼ばれており、この ID を処理するための `dbreg` (“DataBase REGistration”) 関数群を用意してファイル名とログファイル ID のマッピングを管理している。永続版のマッピング (レコードタイプ `DBREG_REGISTER`) がログレコードに書き込まれるのは、データベースをオープンしたときである。しかし、それだけではなくこのマッピングのインメモリでの表現も必要だ。トランザクションの異常終了やリカバリーに対応するために、これが必要となる。このマッピングは、どのサブシステムに任せるべきだろうか?

理屈の上では、ファイルとログファイル ID のマッピングは上位レベルの Berkeley DB の機能だろう。どれかのサブシステムに属するような機能ではない。サブシステムは全体像を意識せずに済むように作られるものだ。当初の設計では、このマッピング情報はログ記録用サブシステムのデータ構造にあった。あの頃はそこに置くのが最適だと思っていたんだ。でも、それから何度となく実装のバグを見つけては修正をしているうちに、マッピングの機能はログ記録用サブシステムから切り出された。そしてそれ単体で小さなサブシステムとし、オブジェクト指向のインターフェイスと private なデータ構造を用意した (今思えば、この情報は Berkeley DB の環境情報自体に持たせるべきだった。サブシステムにするべきではなかった)。

設計講座 **13**

どうでもいいバグなどというものはめったにない。ああ、確かにちょっとした typo とかは時々あるだろう。でもふつうは、バグが発生したということは、何をしたいのかを把握せずに間違った実装をした可能性があるということだ。バグを修正するときには、単におかしなところを見つけるだけではいけない。なぜそうなったのか、何か誤解があったのではないかということを探るようにしよう。そうすれば、そのプログラムのアーキテクチャをより深く理解できるようになるし、そもそもの設計自体に根本的な問題があればそれもはっきりするだろう。

4.9　トランザクションマネージャー: Txn

　最後に取り上げるモジュールはトランザクションマネージャーだ。このモジュールは、独立したコンポーネントたちをとりまとめてトランザクショナルな ACID 特性を提供する。トランザクションマネージャーの役割は、トランザクションの開始と終了 (コミットあるいはアボート)・ログマネージャーやバッファマネージャーとの協調によるトランザクションチェックポイントの受け取り・リカバリーのとりまとめである。それぞれについて、順に説明する。

　ジム・グレイは ACID という言葉の生みの親である。これは、トランザクションが提供する機能をまとめた略語だ [Gra81]。原子性 (Atomicity) とは、一つのトランザクションで実行するすべての操作をひとまとめにして扱うという意味である。データベースにすべて反映されるかひとつも反映されないか、そのいずれかしかない。整合性 (Consistency) とは、トランザクションの実行前後がそれぞれ論理的に整合性のある状態になっているという意味である。たとえば、あるアプリケーションでは、すべての従業員が何らかの部署に所属している必要があるものとする。このとき、そうなっているように強制するのが整合性の役割である。独立性 (Isolation) とは、トランザクションの観点からはすべてのトランザクションがシーケンシャルに実行されるという意味である。他のトランザクションと並列で実行されることはない。最後の永続性 (Durability) とは、トランザクションをいったんコミットしたらそれはコミットされたままになるという意味である。一度コミットした内容は、どんな障害があっても消えることはない。

　トランザクションサブシステムは、ACID 特性を強要するために他のサブシステムにも力を借りている。昔ながらの begin・commit・abort 方式でトランザクションを扱い、トランザクションの開始位置を終了位置を区切る。また、準備済みの呼び出し機能も用意しており、これを使えば 2 フェーズコミットを実現できる。2 フェーズコミットとは分散トランザクション環境でトランザクション扱うためのテクニックだが、本章では扱わない。トランザクションの開始 (begin) では、新しいトランザクション ID を割り当ててトランザクションハンドル `DB_TXN` をアプリケーションに返す。トランザクションの確定 (commit) では、コミットログレコードを記録してからログをディスクに書き出させる (ただし、アプリケーション側でディスクに書き出さないよう指定していた場合はその限りではない。後々の永続性を犠牲にしてでもコミット処理を高速化させたいということもある)。これで、その後何かの障害が発生したとしても、確定したトランザクションの内容はそのまま残るようになる。トランザクションの取り消し (abort) は、ログレコードを逆から読んでそのトランザクションでの作業を取得し、ここまでにそのトランザクションで行ったすべての操作を取り消して、データベースをトランザクション開始前の状態に戻す。

チェックポイントの処理

　トランザクションマネージャーには、チェックポイントを受け取る役割もある。チェックポイントを受け取るには、いくつかの方法がある [HR83]。Berkeley DB で使っているのは、

ファジーチェックポイント方式の一種である。基本的に、チェックポイントとはバッファの内容を Mpool からディスクへと書き出すことだ。これは大掛かりな操作になる可能性もあるが、チェックポイント処理の間でも新たなトランザクションを開始できるようにしておくことが大切だ。そうしないと、サービスの停止時間が長くなってしまう。チェックポイント処理を始めるにあたって、Berkeley DB は現在アクティブなトランザクション群を調べて、それらが書き出した中で一番小さな LSN を探す。この LSN が、チェックポイントの LSN になる。次に、トランザクションマネージャーが Mpool に対して、ダーティなバッファをディスクに書き出すよう指示する。この書き込みが、ログのフラッシュ操作の引き金となることだろう。すべてのバッファをきちんとディスクに書き込めたら、チェックポイント LSN を含むチェックポイントレコードをトランザクションマネージャーが書き込む。このレコードが記録された時点で、チェックポイント LSN 以前のログレコードに書かれた操作がすべてディスク上に記録されたことになる。したがって、チェックポイント LSN より前のログレコードは、もはやリカバリー不要である。ここから得られる結論は次の二つだ。まず、チェックポイント LSN より前のログファイルは消してしまってもかまわないということ。そして、リカバリー処理ではチェックポイント LSN 移行のレコードだけを処理すればよいということ。なぜなら、それより前のレコードの内容は既にディスク上に反映されているからである。

注意すべき点は、チェックポイント LSN と実際のチェックポイントレコードの間には多くのログレコードがある可能性があるということだ。しかし、それでも問題ない。というのも、そういったレコードの内容は論理的なチェックポイントより後に発生したものであり、システムに障害が発生した場合にはそれらもリカバリーが必要になるだろうから。

リカバリー

トランザクションに関するあれこれの中で最後のピースとなるのがリカバリー処理だ。リカバリーの目標は、ディスク上のデータベースを (おそらく整合性が崩れてしまっている状態から) 整合性のある状態に戻すことである。Berkeley DB が使っているのは、由緒正しい 2 パス方式で、簡単に言えば「直近のチェックポイント LSN に対して、コミットされていないすべてのトランザクションを取り消し (UNDO)、コミットされていたトランザクションをすべて再現する (REDO)」といったものである。その詳細は、多少込み入っている。

Berkeley DB はまず、ログファイルの ID と実際のデータベースとのマッピングを再構築しなければいけない。それを使ってデータベース上の操作の REDO や UNDO を行うのだ。ログには `DBREG_REGISTER` レコードの完全な歴史が含まれている。しかし、データベースは長期間稼働しているわけだし、その間のログファイルをずっと残しておかなければいけないというのもあまり望ましくない。そこで、もう少し効率的な方法でこのマッピングにアクセスしていく。チェックポイントレコードを書き込む前に、トランザクションマネージャーが一連の `DBREG_REGISTER` レコード群を書き込む。これは、その時点でのログファイル ID とデータベースのマッピングを表すものである。リカバリーの際には、Berkeley DB はこれらのログレコードを使ってファイルのマッピングを再構築する。

リカバリーが始まると、トランザクションマネージャーがログマネージャーの cached_ckp_lsn の値を見て、ログの中での直近のチェックポイントレコードの場所を調べる。このレコードに、チェックポイントの LSN が含まれている。この LSN 移行のリカバーが必要になるわけだが、そのためにはチェックポイントの LSN の時点でのログファイル ID のマッピングを再構築しなければいけない。この情報が含まれているのは、チェックポイントの LSN **より前の**チェックポイントだ。したがって Berkeley DB は、チェックポイントの LSN の直前に発生したチェックポイントのレコードを読まなければいけない。チェックポイントレコードには、そのチェックポイントの LSN だけではなく直前のチェックポイントの LSN も含まれているので、これを使えばよい。リカバリーは最新のチェックポイントから始まり、各チェックポイントレコードの prev_lsn フィールドを使ってチェックポイントレコードをさかのぼっていく。そして、チェックポイントの LSN のひとつ前のチェックポイントに到達した時点で終了する。アルゴリズム的にはこのようになる。

```
ckp_record = read (cached_ckp_lsn)
ckp_lsn = ckp_record.checkpoint_lsn
cur_lsn = ckp_record.my_lsn
while (cur_lsn > ckp_lsn) {
    ckp_record = read (ckp_record.prev_ckp)
    cur_lsn = ckp_record.my_lsn
}
```

先ほどのアルゴリズムで選んだチェックポイントから始まって、ログをシーケンシャルに読みながらリカバリーを進めてログファイル ID のマッピングを再構築する。ログの終端に達した時点で、システムが停止したときとまったく同じマッピングが復元されているはずだ。またこのとき同時に、トランザクションのコミットのレコードがあればそのトランザクション ID も記録しておく。ログレコードの中に登場するトランザクションの中でそのトランザクション ID がコミットレコードに登場しないものがあれば、それはアボートされたか完了しなかったものとみなす。そのトランザクションは、リカバリー処理でもアボートされたものとして扱わなければいけない。リカバリーがログの終端まで進んだら、向きを反転させて逆向きにログを読んでいく。トランザクションログレコードが見つかるたびにトランザクション ID を取り出し、そのトランザクションがコミットされたものかどうかを調べてレコードの扱い方を判断する。取り出したトランザクション ID がコミット済みトランザクションのリストに存在しない場合は、さらにレコードタイプを調べてそのログレコード用のリカバリールーチンを呼び、ログに記録された操作を UNDO する。取り出したトランザクション ID がコミット済みのものであった場合は、ここでは何もせずそのまま流す。この逆向きのパスを、チェックポイント LSN に戻るまで続ける[1]。最後に、再び方向転換して順方向にログを読む。このときに、コミット済みのトランザクションに属するすべてのログレコードを REDO していく。この最後のパスが完了したら、チェックポイントを作成する。これでデータベースは完全に整合性のある状態になり、アプリケーションを実行する準備が整った。

リカバリー処理をまとめると、次のようになる。

[1] ここではチェックポイント LSN まで戻れば十分であり、そのひとつ前のチェックポイントレコードまで戻る必要はないことに注意

1. 直近のチェックポイントのチェックポイント LSN のひとつ前のチェックポイントを探す
2. そこから順方向に読み進め、ログファイル ID のマッピングとコミット済みトランザクションのリストを構築する
3. チェックポイント LSN に向かって逆方向に読み進め、コミットしていないトランザクション上の操作をすべて UNDO する
4. 順方向に読み進め、コミット済みトランザクション上の操作をすべて REDO する
5. チェックポイントを作る

理屈の上では、最後のチェックポイント処理は不要である。しかし実用性を考慮して、今後のリカバリー処理の時間を短縮したりデータベースを整合性のある状態にしたりするためにチェックポイント処理をしている。

設計講座 **14**

データベースのリカバリーは複雑なトピックであり、実装するのも難しいしデバッグも困難だ。というのも、リカバリー処理はそんなに頻繁に発生するものではないからである。エドガー・ダイクストラは、チューリング賞受賞講演で「プログラミングとは本質的に難しいものであり、そのとてつもない難しさを真摯に認めることからすべてが始まる」と主張した。アーキテクトとして、そしてプログラマーとしての私たちの目標は、さまざまな道具を自由に使いこなすことだ。設計・問題の分割・レビュー・テスト・命名規約やコーディングスタイルなどの道具を使いこなし、プログラミングに関するさまざまな問題を、自分たちで解決**可能**な問題に落とし込む。

4.10　まとめ

Berkeley DB が誕生してから既に 20 年以上が経過した。Berkeley DB は史上初の汎用的かつトランザクション対応なキーバリューストアであり、昨今の NoSQL ムーブメントの始祖だと言っても過言ではないだろう。Berkeley DB は、今でも何百もの商用製品のストレージシステムとして使われており、また何千ものオープンソースアプリケーション (SQL エンジンや XML エンジン、NoSQL エンジンなど) でも同様に使われている。そして、世界中で数え切れないほどの環境にデプロイされている。我々がその開発や保守を通じて学んできたことがコードに込められており、その設計時のちょっとした小技については本章でまとめた。他のソフトウェアデザイナーやアーキテクトにも、ぜひこれらを活用してもらいたいものだ。

第5章

CMake

Bill Hoffman and Kenneth Martin

1999年、米国医療図書館はKitwareという小さな会社と契約した。各種のプラットフォーム上にまたがる複雑なソフトウェアの構成やビルド、そしてデプロイをうまくやる仕組みを開発するためである。この作業は、Insight Segmentation and Registration Toolkit(ITK)[1] の一部であった。プロジェクトのエンジニアリングを率いるKitwareの役割は、ITKの研究者や開発者が使えるようなビルドシステムの開発だった。このシステムは使いやすくなければならず、研究者がプログラミングをするときに最も効率的に使えるようにしなければいけなかった。そんな指令のもとで生まれたCMakeは、年代物のautoconf/libtoolによる手法を置き換えるためのツールである。既存のツールの弱みを克服しつつ、その利点だけは維持するように作られている。

ビルドシステム以外にも、年月を経てCMakeは、各種開発ツールのファミリーへと進化した。CMake、CTest、CPack、そしてCDashだ。CMakeはビルドツールで、ソフトウェアのビルドに利用する。CTestはテストドライバーで、回帰テストを実行するときに使う。CPackはパッケージングツールで、CMakeでビルドしたソフトウェアに各種プラットフォーム用のインストーラーを作るときに使う。CDashはウェブアプリケーションで、テスト結果を表示したり継続的インテグレーションを実行したりする。

5.1 CMakeの歴史と要件

CMakeの開発が始まったころに一般の開発プロジェクトで使われていた手段は、Unix系ならconfigureスクリプトとMakefileだしWindowsならVisual Studioのプロジェクトだった。このように二通りのビルドシステムがあるおかげで、クロスプラットフォームの開発はとても面倒くさいものになっていた。新しいソースファイルをプロジェクトに追加するだけのことでさえ、手間のかかる作業だった。当然、開発者にとってはビルドシステムが統一された

[1] http://www.itk.org/

ほうがありがたい。CMake の開発では、二通りの手法で統一ビルドシステムの問題に立ち向かった。

そのひとつが、1999 年の VTK ビルドシステムだ。このシステムは、Unix 用の configure スクリプトと Windows 用の実行ファイル pcmaker で構成されている。pcmaker は C で書かれたプログラムで、Unix の Makefile を読み込んで Windows 用の NMake ファイルを出力する。pcmaker の実行ファイルは VTK CVS システムリポジトリにチェックインされていた。新しいライブラリを追加するなどのありがちな作業をするにも、ソースを変更して新しいバイナリをチェックインする必要があった。ある意味では統一システムと言えなくもなかったが、問題も多かった。

開発者が使ったもうひとつの手法は、TargetJr 用の gmake ベースのビルドシステムだった。TargetJr は C++で書かれた画像処理環境で、当初は Sun のワークステーション上で開発されていた。そのころの TargetJr は、imake システムで Makefile を作っていた。Windows への移植が必要になったときに作られたのが gmake システムだ。Unix のコンパイラも Windows のコンパイラも、どちらもこの gmake ベースのシステムを使える。このシステムでは、gmake を実行する前に環境変数の設定が必要となる。環境変数を正しく設定しないと実行に失敗し、その原因を突き止めるのは (特にエンドユーザーにとっては) とても難しい。

どちらのシステムにも深刻な問題があった。Windows の開発者にコマンドラインを使わせる必要があるということだ。Windows の開発者は IDE(統合開発環境) を使いたがることが多い。その結果どうなったかというと、Windows の開発者は IDE 用のファイルを手で書いてそれをプロジェクトに追加するようになった。2 系統のビルドシステムが存在する時代に逆戻りだ。IDE 対応の問題以外にも、先述のふたつのシステムには複数のプロジェクトを組み合わせるのがとてつもなく難しいという問題もあった。たとえば VTK (第 24 章) には、画像を読み込むモジュールが決定的に欠けていた。このシステムでは libtiff や libjpeg といったライブラリを使うのがとても難しかったのだ。

そこで、ITK と C++用の汎用的なビルドシステムを新たに開発することになった。新たに作るシステムに必要とされた要件は、次のようなものだ。

- システムにインストールされている C++コンパイラだけに依存すること。
- Visual Studio IDE 用のファイルを生成できること。
- 基本的なビルドシステムターゲット (スタティックライブラリ・共有ライブラリ・実行ファイル・プラグインなど) を簡単に作れること。
- ビルド時にコードジェネレーターを実行できること。
- ひとつのソースツリー内で複数のビルドツリーに分割できること。
- システムのイントロスペクションができること。つまり、ターゲットシステム上で何ができて何ができないのかを自動判別できること。
- C/C++ヘッダファイルの依存関係を自動的にスキャンできること。
- すべての機能が、どのプラットフォームでも同じように安定して動作すること。

外部のライブラリやパーサーへの依存を回避するため、CMake は C++コンパイラだけに依存するように設計した (C++のコードをビルドするなら C++コンパイラは必ずあるだろうとい

う前提だ)。その当時は、Tcl のようなスクリプト言語の環境を UNIX や Windows 上に用意するのは簡単ではなかったのだ。また、現代であっても、モダンなスーパーコンピューターやインターネットから隔離されたセキュアなコンピューターなど、サードパーティのライブラリをビルドしづらい環境は存在する。ビルドシステムはパッケージの基本要件なので、CMake にはコンパイラ以外の依存関係を持ち込まないように決めたのだ。そのせいで CMake 用のシンプルな言語を作ることができなくて、それが原因で CMake を嫌っている人もいる。しかし、その当時最もよく使われていた組み込み用言語が Tcl であったことを思い出そう。もしあのとき CMake が Tcl ベースのビルドシステムを作っていたら、これほどまでに広まることがあっただろうか？

IDE 用のプロジェクトファイルの生成機能は、CMake の強いセールスポイントとなる。しかし同時に、CMake で提供できる機能が IDE がネイティブサポートする機能だけに制限されてしまう。ただ、そんな制限よりも IDE 用のビルドファイルを提供できる利点のほうがずっと上回る。この方針にしたおかげで CMake 自体の開発は難しくなったが、ITK やその他のプロジェクトでの CMake を使った開発がとても簡単になった。開発者にとっては、使いなれたツールを使えるというのはありがたいことだし生産性も上がるというものだ。開発者が自分の好みのツールを使えるようにしたおかげで、開発プロジェクトで最も大切なリソースである「開発者」が大きなアドバンテージを得られるようになった。

すべての C/C++プログラムは、実行ファイルかスタティックライブラリ、共有ライブラリ、プラグインのいずれかをビルドすることになる。CMake の機能を使えば、これらのプロダクトをすべての対応プラットフォーム上で作成できる。すべてのプラットフォームがこれらのプロダクトの作成に対応しているとはいえ、そのときに使うコンパイルフラグはコンパイラによって異なるし、プラットフォームによっても異なる。その複雑性やプラットフォーム間の差異を CMake のシンプルなコマンドの裏側に隠すことで、開発者は Windows や Unix そして Mac 上での開発ができるようになる。プロジェクトそのものの開発に注力でき、共有ライブラリをビルドする方法などの細かいことは意識せずに済むのだ。

コードジェネレータを搭載しようとすると、ビルドシステムはさらに複雑になる。VTK には当初から、C++のコードを自動的にラップして Tcl や Python そして Java で使えるようにするシステムがあった。これは、C++のヘッダーファイルを解析してラッピングレイヤーを自動生成するものである。そのためには、C/C++の実行ファイルをビルドできるビルドシステム (ラッパージェネレータ) が必要となり、ビルド時にそれを実行して C/C++のソースコード (何らかのモジュール用のラッパー) を生成することになる。そして、生成されたソースコードをコンパイルして実行ファイルなり共有ライブラリなりを生成しなければいけない。これらすべてを、IDE 環境や自動生成された Makefile で行う必要があるのだ。

柔軟なクロスプラットフォーム対応の C/C++ソフトウェアを開発する際に重要なのは、システムの機能をプログラムするのであって特定のシステムをプログラムするのではないということだ。Autotools がシステムの状態を把握するために使っているモデルは、ちょっとしたコード片をコンパイルし、その結果を調べて保存しておくというものだ。CMake はクロスプラットフォーム対応を目指しているので、同様のテクニックでシステムの状態を調べること

にした。これで開発者は、個々のシステムへの対応を気にせずシステム本来の機能のプログラムができるようになる。これは、将来のポータビリティを確保するためにも重要だ。というのも、コンパイラや OS は時とともに変わっていくものだからである。たとえば、こんなコードを考えてみよう。

```
#ifdef linux
// Linux 関連の処理
#endif
```

このコードよりも、次のコードのほうが崩れにくい。

```
#ifdef HAS_FEATURE
// 何かの機能を使った処理
#endif
```

初期の CMake の要件には、autotools から引き継いだものがもうひとつあった。それが、ソースツリーとは別にビルドツリーを作れる機能である。この機能を使えば、同じソースツリーに複数のビルド形式を構築できる。また、ソースツリーとビルドファイルがごちゃごちゃになってしまうことも防げる。ごちゃごちゃになってしまえば、バージョン管理システムも混乱してしまう。

ビルドシステムで最も重要な機能のひとつが、依存関係の管理である。あるソースファイルを変更したら、そのソースファイルを使っているすべてのプロダクトを再ビルドしなければならない。C/C++のコードなら、.c ファイルあるいは.cpp ファイルからインクルードしているヘッダファイルの変更もチェックする必要がある。本来コンパイルすべきコードの中にコンパイル漏れがあったせいで依存関係の情報がおかしくなったなどという問題の原因を追うのは時間の無駄だ。

新しく作るビルドシステムは、これまでにあげたすべての要件や機能をすべての対応プラットフォームで同様に実現する必要があった。CMake に求められたのは、開発者が複雑なソフトウェアシステムを作るときにプラットフォームの詳細を知らなくても済むようなシンプルな API だった。事実上、CMake を使っているソフトウェアはビルドに関する複雑なあれこれを CMake チームにアウトソースしているに等しい。これらの基本要件に基づいたビジョンができあがったら、その実装をアジャイル手法で進めていくなければばいけなかった。というのも、ITK は新しいビルドシステムをすぐにでも必要としていたのだ。最初のバージョンの CMake はここまでにあげた要件をすべて満たしているわけではなかったが、Windows と Unix 上でのビルドはできるものだった。

5.2　CMake の実装方法

先述の通り、CMake 自体の開発に使った言語は C と C++である。その内部を解説するにあたって、このセクションではまず CMake の動きをユーザーの視点から説明する。それから、その構造を見ていこう。

CMake の処理工程

CMake には、主要なフェーズが二つある。まず最初が “構成 (configure)” で、ここでは CMake がすべての入力を処理し、ビルドを実行するための内部表現を生成する。次のフェーズが “生成 (generate)” で、実際のビルドファイルをここで生成する。

環境変数 (またはそれ以外の変数)

1999 年当時に限らず現在でも、プロジェクトのビルドの際にシェルの環境変数が使われている。よくありがちな使い方は、環境変数 PROJECT_ROOT でソースツリーのルートの場所を表すといったものだ。環境変数は、オプションパッケージや外部パッケージの場所を表すときにも用いられる。この手法の問題は、ビルドのたびにこれらの環境変数を設定しないとビルドがうまくいかないということだ。この問題を解決するために、CMake はキャッシュファイルを作っている。このキャッシュファイルに、ビルドに必要なすべての変数を格納しているのだ。シェルの環境変数などではなく、CMake 独自の変数である。あるビルドツリー上で最初に CMake を実行したときに `CMakeCache.txt` というファイルを作って、そこにビルド用の変数をすべて格納する。このファイル自体もビルドツリーの一員なので、CMake を実行するたびにその内容を参照できる。

構成ステップ

構成ステップでは、まず最初に前回実行したときの `CMakeCache.txt` があるかどうかを調べ、存在すればそれを読み込む。それに続いて、ソースツリーのルートにある `CMakeLists.txt` を読み込む。構成ステップで `CMakeLists.txt` をパースするのは CMake 言語パーサーだ。このファイル内で見つかった CMake コマンドを、コマンドパターンオブジェクトが実行する。このステップでさらに別の `CMakeLists.txt` ファイルをパースさせることもできる。そのために使える CMake のコマンドが `include` と `add_subdirectory` だ。CMake は個々のコマンドに対応する C++のオブジェクトを持っており、これを CMake 言語から使えるようになっている。たとえば `add_library` や `if`、`add_executable`、`add_subdirectory`、そして `include` といったコマンドがある。事実上、CMake 言語はこれらのコマンドの呼び出しで実装されている。パーサーは、単純に CMake の入力ファイルをコマンド呼び出しに変換するだけであり、文字列のリストはそれらのコマンドへの引数となる。

構成ステップは本質的に、ユーザーが指定した CMake のコードを “実行” するステップだ。すべてのコードを実行してキャッシュ変数の値を算出し終えたら、CMake はそのプロジェクトのインメモリ表現を得たことになる。この中にはすべてのライブラリや実行ファイルそしてカスタムコマンドが含まれており、選んだジェネレータで最終的にビルドするために必要な情報がすべて含まれている。この時点で `CMakeCache.txt` ファイルを保存し、次に CMake を実行したときに使えるようにする。

プロジェクトのインメモリ表現とはターゲットの集まりである。ターゲットとは単にビルド対象のものを指す。ライブラリや実行ファイルなどである。CMake はカスタムターゲットもサポートしている。ユーザーが自分で入出力を定義し、ビルド時に実行するカスタムコマンドやスクリプトを用意すればよい。CMake は、各ターゲットを `cmTarget` オブジェクトに格納する。そして、これらのオブジェクトがさらに `cmMakefile` オブジェクトに格納される。`cmMakefile` オブジェクトは基本的に、ソースツリー内の指定したディレクトリにあるすべてのターゲットを格納するものである。最終的にできあがるのは `cmMakefile` オブジェクトのツリーで、その中に `cmTarget` オブジェクトのマップが含まれている。

生成ステップ

構成ステップが完了したら、次は生成ステップの出番だ。このステップは、ユーザーが選んだターゲットビルドツール用のビルドファイルを CMake が生成する。この時点で、ターゲット (ライブラリ・実行ファイル・カスタムターゲット) の内部表現を変換して、Visual Studio などの IDE 用のファイルあるいは `make` で実行するための Makefile 群にする。構成ステップを終えた時点での CMake の内部表現は可能な限り汎用的な形式にしており、異なるビルドツール間でもできるだけ多くのコードやデータ構造を共有できるようにしている。

これらの工程の概要を図 5.1 に示す。

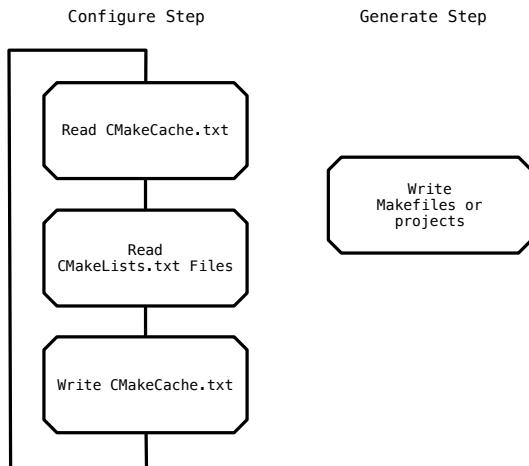



図 5.1: CMake の処理工程の概要

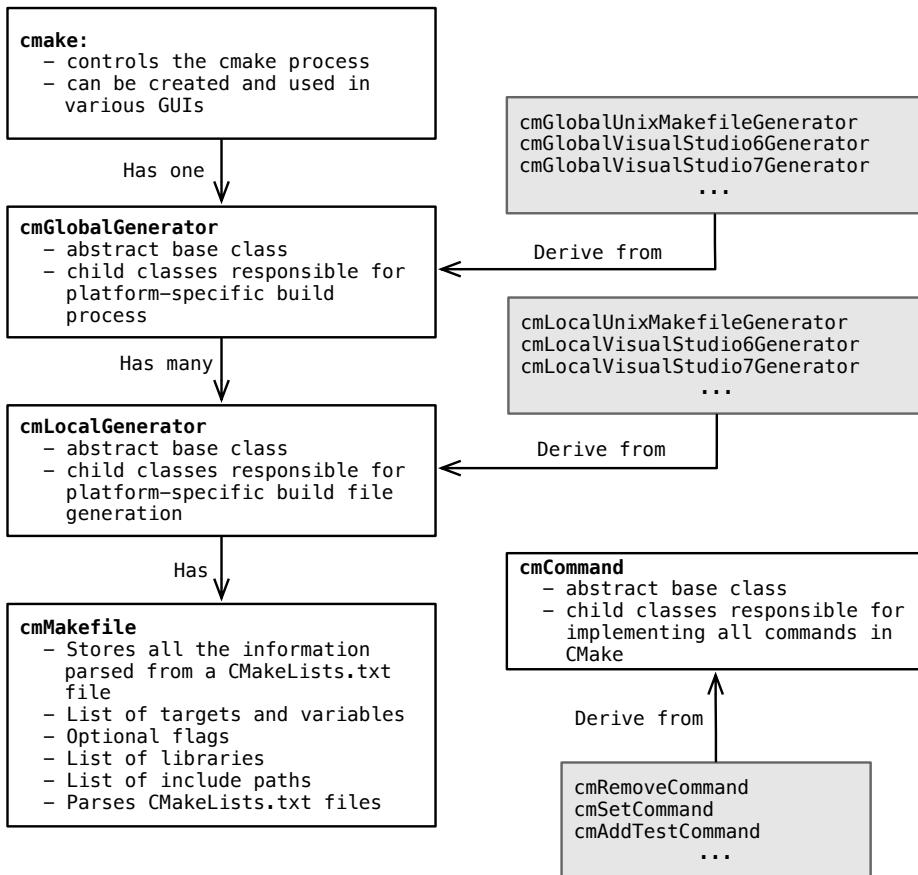



図 5.2: CMake のオブジェクト群

CMake: そのコード

CMake のオブジェクト群

CMake はオブジェクト指向のシステムで、継承やデザインパターンそしてカプセル化などのテクニックを駆使している。主要な C++オブジェクトとその間の関連を図 5.2 に示す。

各所の `CMakeLists.txt` ファイルをパースした結果は `cmMakefile` オブジェクトに格納する。そのディレクトリに関する情報を保持するだけでなく、`cmMakefile` オブジェクトは `CMakeLists.txt` ファイルのパースも担当する。パース関数がこのオブジェクトを呼び、このオブジェクトが lex/yacc ベースのパーサで CMake 言語を処理する。CMake 言語構文が変わることはほとんどないし、CMake 自体をビルドするシステム上で lex や yacc が常に使えるとは限らない。そこで、事前に lex と yacc で処理したファイルもバージョン管理システムの `Source` ディレクトリに格納している。

CMake で重要なもう一つのクラスが cmCommand である。これは、CMake 言語の全コマンドの実装時に基底クラスとして用いるものだ。各サブクラスにはコマンドの実装だけでなくそのドキュメントも含まれている。例として、cmUnsetCommand クラスのドキュメント用メソッドを紹介する。

```
virtual const char* GetTerseDocumentation()
{
    return "Unset a variable, cache variable, or environment variable.";
}

/**
 * More documentation.
 */

virtual const char* GetFullDocumentation()
{
    return
      "  unset(<variable> [CACHE])\n"
      "Removes the specified variable causing it to become undefined.  "
      "If CACHE is present then the variable is removed from the cache "
      "instead of the current scope.\n"
      "<variable> can be an environment variable such as:\n"
      "  unset(ENV{LD_LIBRARY_PATH})\n"
      "in which case the variable will be removed from the current "
      "environment.";
}
```

依存関係の解析

CMake には強力な依存関係解析機能が組み込まれており、Fortran や C そして C++ のソースファイルを解析できる。統合開発環境 (IDE) は自前でファイルの依存関係を管理しているので、IDE 向けのビルドの場合は CMake での依存関係解析処理をスキップする。IDE 向けのビルドでは、CMake は IDE ネイティブの入力ファイルを生成し、ファイルレベルの依存関係情報は IDE に処理させる。ターゲットレベルの依存関係情報は、IDE 向けのフォーマットに変換して指定する。

Makefile ベースのビルドの場合、make 自体には依存関係を最新の状態に保つ機能がない。そこで、この場合は CMake が自動的に C・C++・Fortran のファイルの依存関係を算出する。依存情報の生成や最新の状態への追従は、CMake が自動的に行うということだ。プロジェクトを最初に CMake で構成すれば、あとは単に `make` を実行するだけでよい。残りの作業はすべて CMake がやってくれる。

ユーザーは CMake がどのように動いているかなど知る必要もないが、プロジェクトの依存情報のファイルがどのようになっているかは見る価値があるだろう。各ターゲット用の依存情報は `depend.make`、`flags.make`、`build.make`、そして `DependInfo.cmake` の 4 つのファイルに格納されている。`depend.make` には、ディレクトリ内のすべてのオブジェクトファイルの依存情報を保存する。`flags.make` には、このターゲットのソースファイルに使うコンパイルフラグを保存する。この内容が変わったら、ファイルが再コンパイルされる。`DependInfo.cmake` は、依存情報を最新に保つために利用する。また、プロジェクトに属するファイルがどれでそこにどんな言語を使っているのかといった情報もこのファイルに含まれる。最後に、依存

関係を構築するためのルールが build.make に保存される。あるターゲット用の依存情報が古くなるとそのターゲット用の依存情報を再度算出し、依存情報を最新の状態に保つ。その理由は、.h ファイルに変更があれば新たな依存関係が増える可能性があるからである。

CTest および CPack

現在に至るまでに CMake は成長を続け、単なるビルドシステムではなくビルド・テスト・パッケージングなどのツールをまとめたシステムになった。コマンドラインツールである cmake や GUI プログラムの CMake に加えて、テスティングツールである CTest やパッケージングツールである CPack なども一緒に配布されている。CTest や CPack は CMake と同じコードベースを共有しているが CMake とは別のツールであり、基本的なビルドをする上では必須ではない。

実行ファイル ctest を使えば回帰テストを実行できる。CTest 用のテストを作るのも簡単で、add_test コマンドを使えばよい。テストを実行するときに CTest も使え、これを使ってテスト結果を CDash アプリケーションに送ればウェブで確認できるようになる。CTest と CDash の組み合わせは、テスティングツールである Hudson と似ている。が、大きく異なる点がひとつある。CTest は、より分散したテスト環境を構築できるように作られている。クライアントがバージョン管理システムからソースを取得してテストを実行し、その結果を CDash に送信するといったものだ。Hudson の場合は、クライアントマシンへの ssh アクセス権を Hudson に渡さなければテストを実行できない。

実行ファイル cpack を使うと、プロジェクトのインストーラーを作れる。CPack の動きは CMake のビルド部と似ている。他のパッケージングツールとの橋渡し役となるのだ。たとえば Windows の場合は、パッケージングツール NSIS を使ってプロジェクトのインストーラーを作る。CPack はプロジェクトのインストールルールに基づいてインストールツリーを作り、これを NSIS のようなインストーラー作成プログラムに渡す。CPack がその他に対応しているのは、RPM や Debian .deb ファイル、.tar、.tar.gz そして自己展開形式の tar ファイルである。

グラフィカルインターフェイス

多くのユーザーにとって、CMake で最初に目にするのは CMake のユーザーインターフェイスプログラムだ。CMake には二種類のインターフェイスがある。Qt ベースのウィンドウアプリケーションとコマンドラインの curses ベースのアプリケーションだ。これらの GUI は、CMakeCache.txt ファイル用のグラフィカルなエディタである。インターフェイスは比較的シンプルで、configure(構成) と generate(生成) の二つのブタンがあるだけである。これらを使って、CMake のそれぞれの工程を実行する。curse ベースの GUI は、Unix 系の TTY プラットフォームや Cygwin で使える。Qt の GUI は、すべてのプラットフォームで使える。それぞれの GUI の様子を図 5.3 と図 5.4 に示す。

図 5.3: コマンドラインのインターフェイス

図 5.4: グラフィックベースのインターフェイス

どちらのインターフェイスでも、キャッシュ変数の名前が左側にあってその値が右側にある。右側の値は、ユーザーが適切な値に変更できる。変数には、ノーマルとアドバンストの二種類がある。デフォルトでは、ノーマル変数がユーザーに表示される。どの変数がノーマルでどの変数がアドバンストかは、そのプロジェクトの `CMakeLists.txt` ファイルの中で決める。これで、ビルドに必要な変数だけをユーザーに見せるようにできる。

コマンドを実行するとキャッシュ変数の値が書き変わるので、最終的なビルドをとりまとめる処理は漸進的に行える。たとえば、あるオプションを有効にするまで別の追加オプションは見えないといった具合だ。そのため、ユーザーがすべてのオプションを少なくとも一度は見るまでは GUI の "generate" ボタンを無効にしている。configure ボタンを押すたびに、まだユーザーに見えていないキャッシュ変数が赤く表示される。赤く表示されるキャッシュ変数がなくなった時点で generate ボタンが使えるようになる。

CMake のテスト

CMake 自体の開発に新たに加わった人が最初に見せられるのが、CMake の開発におけるテストの手順である。ここでは CMake ファミリーのツール群 (CMake、CTest、CPack そして CDash) を使っている。コードを書いてバージョン管理システムにチェックインすると、継続的インテグレーション用のテストマシンが自動的に新たな CMake のビルドとテストを (CTest を使って) 行う。その結果は CDash サーバーに送られ、もしビルドエラーやコンパイル時の警告あるいはテストの失敗などが発生していれば開発者にメールを送信する。

一連の流れは、典型的な継続的インテグレーション環境と同じだ。CMake のリポジトリに新しいコードがチェックインされたら CMake がサポートする各種プラットフォーム上でのテストが自動的に行われる。CMake はさまざまなコンパイラやプラットフォームに対応しているので、安定したビルドシステムの開発にはこの種のテストシステムが必要となる。

たとえば、誰かが新たなプラットフォームをサポートしたいと思ったとしよう。その人が最初に聞かれるのは、そのプラットフォーム用の日々のダッシュボードクライアントを用意できるのかどうかということだ。継続的なテストができなければ、新たなシステムのサポートなど不可能だ。

5.3 教訓

CMake は ITK の開発初日から無事に動き出した。そして、そのプロジェクトで最も重要な位置を占めるようになった。もし仮に CMake を一から作り直すことになったとしても、今と大きくは変えないだろう。とはいえ、こうしておけばよかったということがないわけではない。

後方互換性

後方互換性を維持することは、CMake 開発チームにとって重要なことだった。このプロジェクトの第一の目標は、ソフトウェアのビルドを容易にすることである。どこかのプロジェクトや開発者がビルドツールとして CMake を採用したとして、そのときに重要なのは、CMake のバージョンを上げげたとたんにビルドできなくなるなどという事態を何としてでも避けるということだ。CMake 2.6 ではポリシーシステムを導入した。これは、CMake の既存の振る舞いを変えてしまうような変更をすると警告を発して今までの振る舞いをし続けるというものだ。これを使うには、使おうとしている CMake のバージョンを `CMakeLists.txt` ファイルで指定しておく必要がある。新しいバージョンの CMake でビルドすると警告が発生するが、古いバージョンのときと同じ動きでビルドが進む。

言語、げんご、ゲンゴ

CMake 言語は、とにかくシンプルなものにするという方針で作った。しかしそれが、新たなプロジェクトで CMake の採用を検討するにあたっての大きな障害になることが多かった。現在の CMake 言語には、いくつかおかしな動きがある。最初のパーサーは lex/yacc すら使っておらず、単に文字列をパースするだけのものだった。CMake 言語について考え直すチャンスがあれば、既に存在するよくできた組み込み言語を調査しただろう。一番相性がよさそうなのは、きっと Lua だ。コンパクトにまとまっていて、すっきりとしている。Lua のような外部の言語を使わないにしても、既存の言語をもっと早いうちから調べるべきだった。

プラグインが動かない!

CMake 言語をプロジェクトごとに拡張できるように、プラグインクラスを用意している。これを使えば、各プロジェクト用の新たな CMake コマンドを C で作れる。この機能を作った当時はよさげなアイデアに思えたし、インターフェイスは C 用に定義したのでさまざまなコンパイラで使えた。しかし、32 ビット/64 ビットの Windows や Linux などのさまざまな API システムが登場し、プラグインの互換性を維持するのが難しくなってきた。プラグインを使わずに CMake の言語だけで CMake を拡張するのはあまり強力な仕組みではないが、プラグインのビルドや読み込みに失敗したせいで CMake がクラッシュしたりプロジェクトがビルドできなかったりといったことは回避できる。

公開 API の削減

CMake の開発で得た大きな教訓は、ユーザーが使いもしない機能の後方互換性をわざわざ維持する必要などないということだ。CMake の開発中に何度か、ユーザーや顧客から「CMake 自体をライブラリ化して他の言語からその機能を使えるようにしてほしい」という要望を受けた。そんなことをすれば、いろんな使い方をする人たちが出てきて CMake のユーザーコミュニティが分断されるし、さらに CMake プロジェクトのメンテナンスコストも増大してしまう。

第6章

継続的インテグレーション

C. Titus Brown and Rosangela Canino-Koning

継続的インテグレーション (Continuous Integration: CI) システムとは、ソフトウェアのビルドやテストを自動的かつ定期的に行うシステムのことである。CI システムを使う最大のメリットは、ビルドやテストの実行間隔が長くなるのを避けられるということだ。それ以外にも、その他の退屈な作業を単純化して自動化させることもできる。たとえば、クロスプラットフォームでのテスト実行、遅かったりデータを扱ったり設定の難しかったりするテストの定期実行、レガシーな環境での適切なパフォーマンスの確保、ごくまれに失敗するテストの検出、そしてリリースする製品の定期的な作成などといった作業がそれにあたる。また、ビルドやテストの自動化は継続的インテグレーションのために必須となるので、CI は**継続的デプロイ**用フレームワークに向けた第一歩にもなる。これは、ソフトウェアを更新したら、テストをしてすぐに稼働中のシステムへ展開できるようにする仕組みである。

昨今アジャイルソフトウェア方法論が広まってきたこともあり、継続的インテグレーションはタイムリーな話題である。オープンソースの CI ツールもここ数年急増し、さまざまな言語向けにさまざまな言語で書かれている。そして、さまざまなアーキテクチャモデルに対応した幅広い機能が実装されている。本章では継続的インテグレーションシステムが実装する一般的な機能群について説明し、アーキテクチャに関する選択肢について議論する。そして、選んだアーキテクチャごとにどの機能が実装しやすくてどの機能が実装しづらいのかを検討する。

これ以降では、CI システムを設計する際に選択可能なアーキテクチャのよい例となるシステム群について簡単に説明する。最初に取り上げる Buildbot はマスター/スレーブ型のシステムだ。それに続く CDash はレポートサーバー型、Jenkins はハイブリッド型、そして最後の Pony-Build は Python ベースの分散型レポートサーバーで、これを使ってさらに議論を深めていく。

6.1 概観

継続的インテグレーションシステムのアーキテクチャの世界は二大勢力に支配されているようだ。一方はマスター/スレーブ型のアーキテクチャで、中央のサーバーがリモートのビルドを指揮して制御する。その対極にあるのがレポーティングアーキテクチャで、各クライアントからのレポートを中央のサーバーが集約する。我々の知る範囲では、すべての継続的インテグレーションシステムはこの二種類のアーキテクチャの機能を組み合わせて使っている。

中央集権型のアーキテクチャの例として取り上げる Buildbot は、ふたつのパーツで構成されている。中央サーバーである *buildmaster* がそこに接続しているクライアントのビルドスケジュールを管理し、クライアント側の *buildslaves* が実際のビルドを行う。buildmaster はクライアントからの接続先となり、各クライアントがどのコマンドをどの順で実行するのかという設定情報を提供する。buildslave は buildmaster に接続詞、詳細な指示を受け取る。buildslave の設定に含まれるのは、ソフトウェアをインストールすることやマスターサーバーの識別、そしてマスターサーバーに接続するための認証情報などである。ビルド予定をたてるのは buildmaster で、その出力は buildslave から buildmaster に流される。結果はマスターサーバー上に保持され、ウェブ経由で見たり別のレポートシステムや通知システムで見たりすることができる。

アーキテクチャ的にその対極にあるのが CDash で、これは Kitware, Inc. の Visualization Toolkit (VTK)/Insight Toolkit (ITK) プロジェクトで使われている。CDash は本質的にレポーティングサーバーで、CMake および CTest を実行するクライアントコンピューターから受け取った情報を蓄積して表示するように作られている。CDash では、クライアント側がビルドとテストスイートを起動し、ビルドとテストの結果を記録し、それから CDash サーバーに接続して情報をレポーティングサーバーに預ける。

最後に、三番目の例として取り上げる Jenkins (かつては Hudson と呼ばれていたが 2011 年に名前が変わった) は、その両方の操作モードを提供している。Jenkins の場合、ビルドを個別に実行して結果をマスターサーバーに送ることもできるし、ノードをすべて Jenkins マスターサーバーの支配下においてビルドの予定やその実行をマスターサーバーから指示することもできる。

中央集権型モデルと分散型モデルの両方に共通する機能もあり、Jenkins を見てもわかるとおり、両方のモデルをひとつの実装に共存させることもできる。しかし Buildbot と CDash はお互い全く正反対の存在である。ソフトウェアをビルドしてその結果を報告するという点は共通しているが、それ以外の面では全く異なるアーキテクチャを採用している。なぜだろう?

アーキテクチャの選択によって、特定の機能の実装しやすさ (しにくさ) はどの程度の影響を受けるのだろう? 中央集権型を採用することで必然的に出てくる機能などがあるのだろうか? 既存の実装の拡張しやすさについてはどうだろう—レポーティングの仕組みに手軽に手を入れたり、多数のパッケージを扱うために規模を拡大したり、あるいはビルドやテストをクラウド環境で実行したりといったことはできるのだろうか?

継続的インテグレーションソフトウェアの役割は?

継続的インテグレーションシステムの中核となる機能は単純だ。ソフトウェアをビルドしてテストを実行し、その結果を報告するだけである。ビルドやテストそして結果報告はスクリプトで行える。これは、スケジュールを組み込んだタスクや cron ジョブとして実行する。スクリプトの仕事は、ソースコードの新たなコピーを VCS から取得してビルドし、そしてテストを実行することだ。出力はログファイルに書き込むことになるだろう。ファイルを所定の場所に保存し、ビルドが失敗したときにはメールを送信することになる。この機能を実装するのは簡単だ。UNIX なら、大半の Python パッケージについてたった 7 行のスクリプトでこの機能を実現できる。

```
cd /tmp && \
svn checkout http://some.project.url && \
cd project_directory && \
python setup.py build && \
python setup.py test || \
echo build failed | sendmail notification@project.domain
cd /tmp && rm -fr project_directory
```

図 6.1 において影付きでない長方形は、システム内にある個別のサブシステムや機能をを表す。矢印は、コンポーネント間の情報の流れを意味する。雲で囲まれている部分は、おそらくリモートで実行されるであろうビルドプロセスを表す。影付きの長方形は、サブシステム間のつながりを表す。たとえば、ビルドの監視にはビルドプロセス自体の監視とシステムの健康状態 (CPU の負荷、入出力の負荷、メモリの使用量など) の監視が含まれる。

しかし、単純そうに見えるのは見かけだけである。実際の CI システムは、通常はこれ以上のことを行っている。リモートのビルドプロセスを立ち上げてその結果を受け取ったりするだけでなく、継続的インテグレーションソフトウェアはこのような追加機能に対応していることもある。

チェックアウトと更新: 大規模なプロジェクトでは、ソースコードすべてを新たにチェックアウトするのは帯域的にも時間的にもコストがかかることになる。通常、CI システムは既存の作業コピーをその場で更新することになる。更新の際にやりとりするのは、前回の更新以降の差分だけである。通信量は節約できるが、その代わりにシステム側で作業コピーの状況をわかっていなければならない、更新方法も知る必要がある。つまり、通常は少なくとも VCS とは最小限の統合をすることになる。

ビルドレシピの抽象化: 設定やビルドそしてテストのレシピは、対象となるソフトウェア用に書かなければならない。もとになるコマンドは (Mac OS X と Windows と UNIX など)OS によって異なることが多い。ということは、それぞれの OS に特化したレシピを書く (これはバグのもとになるし、実際のビルド環境とはかけ離れてしまう可能性もある) か、さもなければ何らかの抽象化をしてレシピを CI 構成システムから提供できるようにしなければならない。

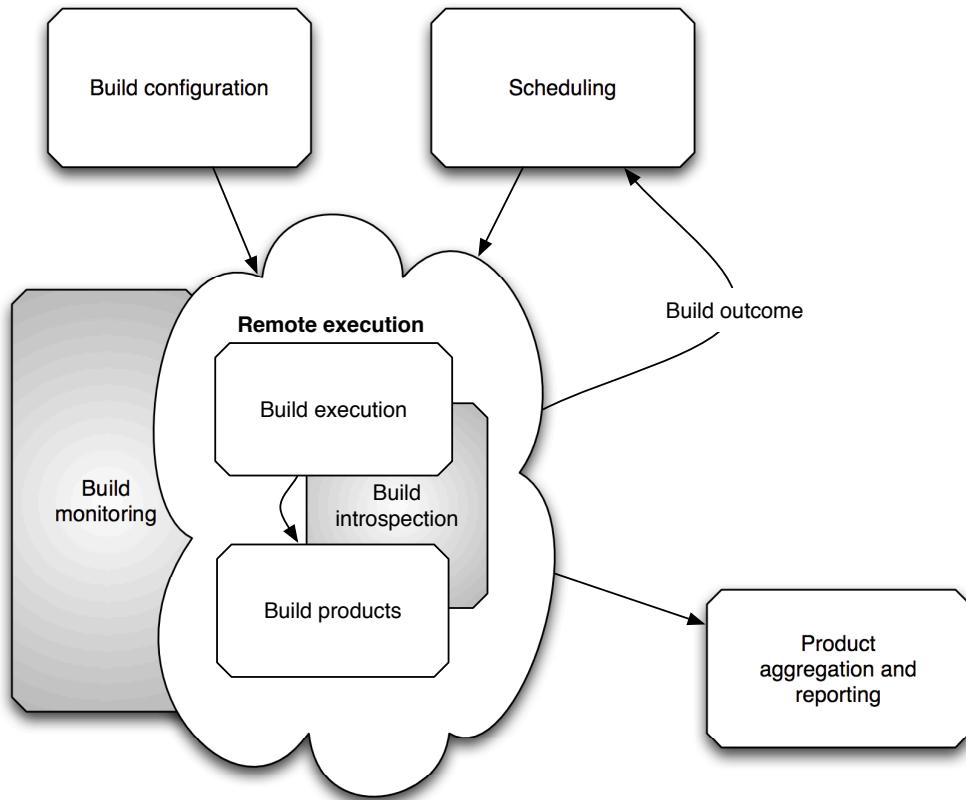



図 6.1: 継続的インテグレーションシステムの内部構造

チェックアウト/ビルド/テスト の状態の保存: チェックアウトの詳細 (更新されたファイル、コードのバージョンなど) やビルドの情報 (警告やエラー)、そしてテストの結果 (コードカバレッジ、パフォーマンス、メモリの使用量) などを保存し、あとで解析に使えるようにしたいという要望もあるだろう。これらの結果を使えば、ビルドアーキテクチャをまたがる質問 (最新のチェックインのせいで特定のアーキテクチャのパフォーマンスに問題が出ていないか?) や歴史を超えた質問 (コードカバレッジは先月に比べて劇的に上昇したか?) にも答えられるようになる。ビルドレシピと同様、この種の調査の仕組みやデータ形式は、プラットフォームやビルドシステムに依存するものとなる。

パッケージのリリース: ビルドを実行するとバイナリパッケージあるいはその他外部に公開する必要のある何かができあがるかもしれない。たとえば、ビルドマシンに直接アクセスできない開発者が、最新のビルドを特定のアーキテクチャでテストしたくなることもあるだろう。これをサポートするためには、CI システムがビルドの成果物を中央リポジトリに転送できるようにしておく必要がある。

複数のアーキテクチャでのビルド: 継続的インテグレーションの目的のひとつは複数のアーキテクチャでビルドしてクロスプラットフォームな機能をテストすることなので、CI ソフトウェアは各ビルドマシンのアーキテクチャを追跡してビルドやビルド結果を各クライアントにリンクしなければならない。

リソース管理: ビルドの手順が特定のマシンのリソースの状況に依存する場合、CI システムはビルドを条件付きで実行させたくなることもある。たとえば、他のビルドやユーザーがいない間はビルドを待ったり、CPU やメモリの使用率が一定に達したらビルドを遅らせたりということがあり得る。

外部リソースとの協調: インテグレーションテストはローカルにないリソースに依存することがある。ステージング環境のデータベースやリモートウェブサービスなどだ。したがって、CI システムは複数のマシン間で協調し、これらのリソースへのアクセスを整理する必要がある。

進捗レポート: 時間がかかるビルド手順については、ビルド状況の定期的な報告も大切である。5 時間におよぶビルドやテストの中で主に知りたいのは最初の 30 分の結果だったとしよう。最後まで実行しないと何も結果を見られないのは時間の無駄になる。

CI システムで必要となりそうな全コンポーネントの概要は図 6.1 に示したとおりだ。CI ソフトウェアは通常、これらのコンポーネントの一部を実装している。

外部とのインタラクション

継続的インテグレーションシステムでは、他のシステムとのやりとりも必要となる。考えうるやりとりには、次のような型がある。

ビルド通知: ビルドの結果は、一般的にクライアントとのやりとりを要するだろう。プル形式での取得 (ウェブ、RSS、RPC など) あるいはプッシュによる通知 (メール、Twitter、PubSubHubbub など) のいずれかとなる。すべてのビルド結果を通知することもあれば失敗したビルドだけを通知することもある。あるいは、所定の時間内に実行できなかったビルドだけを通知することもある。

ビルド情報: ビルドの詳細やその成果物を取得しなければならないこともあるだろう。通常は、RPC を使うか一括ダウンロードの仕組みを用意する。たとえば、別の解析システムを使ってより詳細な (あるいはより的を絞った) 解析を行ったり、コードカバレッジやパフォーマンスの情報を表示させることがある。さらに、テスト結果のリポジトリを別に用意して、CI システムでの失敗したテストと成功したテストの記録を保存しておくこともあるかもしれない。

ビルド要求: ユーザーあるいはコードリポジトリからのビルド要求を受け、それに対応する必要があるかもしれない。大半の VCS にはコミット後に何らかの処理をフックする仕組みがあり、たとえばビルド処理を起動するような RPC コールを実行することができる。あるいは、ユーザーがウェブインターフェイスあるいは RPC を使って手動でビルド要求を出すかもしれない。

CI システムのリモート制御: より一般化して、実行環境全体の変更をある程度うまく作られた RPC インターフェイスで行いたいものだ。アドホックな拡張あるいは正式に決められたインターフェイスを使って、特定のプラットフォーム上でのビルドの実行や別のブランチにさまざまなパッチを適用したビルドの実行、あるいは条件付きでのビルドの実行などを実行できる必要がある。この機能があれば、より一般的なワークフローにも対応できるので便利だ。たとえば CI テストに完全にパスした変更だけをコミットできるようにしたり、パッチをさまざまなシステムでテストしてから最終的に取り込むようにしたりといったことができる。バグ追跡システムやパッチシステムその他外部のシステムにはさまざまなものがあるので、このロジックを CI システム自体に組み込んでしまうのは意味がない。

6.2　アーキテクチャ

Buildbot と CDash はまったく正反対のアーキテクチャを選択しており、一部重複する部分もあるが別々の機能群を実装している。個々の機能セットについて以下で吟味し、ある機能の実装しやすさ (しにくさ) がアーキテクチャの選択でどのように変わるのかを確かめる。

実装モデル: **Buildbot**

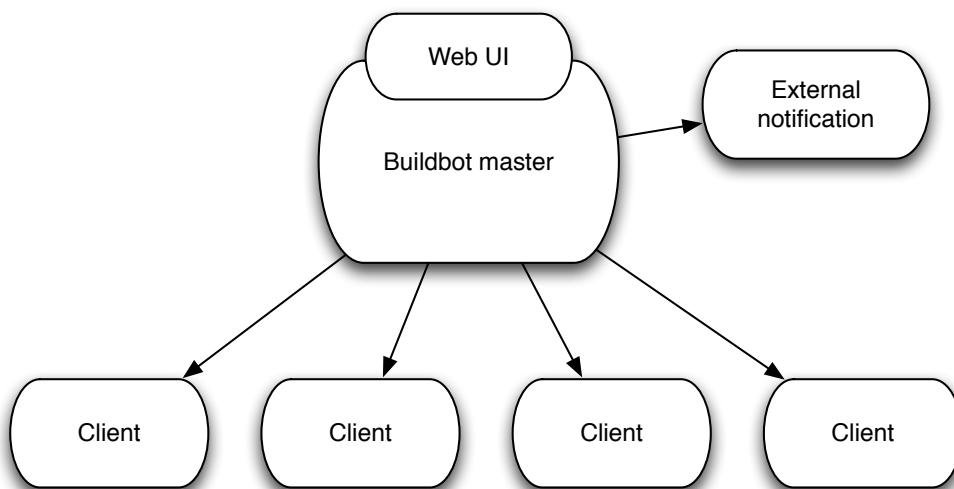



図 6.2: Buildbot のアーキテクチャ

Buildbot はマスター/スレーブ型のアーキテクチャで、一台の中央サーバーと複数台のビルド用スレーブで構成されている。リモートの実行は、完全にマスターサーバー側からの指示

でリアルタイムに行われる。各クライアント上で実行するコマンドをマスター側で設定し、直前のコマンドが終了するとそれを実行する。スケジューリングやビルド要求をサーバー側でとりまとめるだけではなく、実際の指示もマスターが行う。レシピの抽象化機能は組み込まれていない。ただし、基本的なバージョン管理システムとの統合 (“我々のコードはこのリポジトリにある”)、そしてビルドディレクトリ上で実行されるコマンドとビルドディレクトリ内で実行されるコマンドの区別は例外である。OS 固有のコマンドは、通常は設定で直接指定する。

Buildbot は各スレーブとの接続を持続させ、ジョブの管理やスレーブ間での調整を行う。持続的接続を使ったリモートマシンの管理の実装は複雑なものとなり、長年バグの元となり続けている。堅牢なネットワーク接続を長期間保つのは単純なことではなく、ローカルの GUI と対話するアプリケーションのテストをネットワークごしに行うのは大変だ。OS のアラートウィンドウは特に扱いにくいものである。しかし、このように接続を持続させるおかげで、リソースの協調やスケジューリングを直感的に行うことができる。ジョブを実行させるときには、マスターがスレーブを完全に操れるからだ。

Buildbot のモデルの設計に組み込まれたような密な制御は、中央管理型でビルドを管理してリソース間で協調させることが容易にできる。Buildbot は buildmaster 上でのマスターのロックとスレーブのロックの両方を実装している。これらを使って、システムグローバルなリソースとマシンローカルなリソースを協調させたビルドが可能となる。この点で、Buildbot が特に適しているのは大規模なシステムのインテグレーションテスト (データベースやその他高価なリソースと組み合わせたテスト) であると言える。

しかし、中央集権型の設定は、分散型の利用モデルでは問題の原因となる。新しい buildslave を追加するには、マスターの設定で明示的に許可しなければならない。つまり、新しい buildslave を動的に中央サーバーにアタッチしてビルドサービスやビルド結果を送るようにするのは不可能だということだ。さらに、個々のスレーブは完全にマスター側からの指示で動いているので、悪意のある設定をされたり設定を間違えてしまったりするといった事故に対して脆弱になる。クライアント OS のセキュリティ制約の範囲内で、マスターはクライアントを文字通り完全に支配する。

Buildbot の機能面での制限のひとつは、ビルドの成果物を中央サーバーに返すシンプルな方法がないことだ。たとえば、コードカバレッジの統計情報やビルド後のバイナリはリモートの buildslave 上に残ったままとなる。中央の buildmaster 側に、それを集約したり配布したりする API は用意されていない。この機能が存在しない理由は定かではない。Buildbot とともに配布されているコマンド群の抽象化の制約のためかもしれない。もとのコマンド群が、スレーブ上でのリモートコマンドの実行に焦点を合わせたものだからだ。あるいは、buildmaster と buildslave との間の接続はあくまでも制御システムとして使い、RPC の仕組みとしては使わないように決めた結果かもしれない。

マスター/スレーブモデルを採用し、かつこのように制限のある通信チャンネルを用意した結果、buildslave 側からはシステムの利用状況を報告できなくなっており、マスター側ではスレーブの負荷対策を組み込むことができない。

ビルド結果の外部 CPU 通知は完全に buildmaster が処理する。新たな通知サービスを追加するには buildmaster 自身の中で実装しなければならない。同様に、新たなビルド要求は直接 buildmaster にしなければならない。

実装モデル: CDash

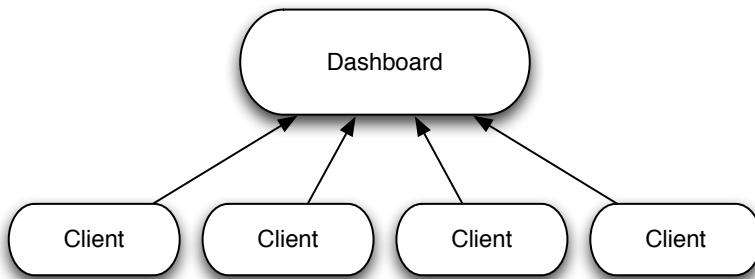



図 6.3: CDash のアーキテクチャ

Buildbot とは対照的に、CDash はレポーティングサーバーモデルを実装している。このモデルにおいて、CDash サーバーは中央リポジトリとしてふるまう。リモートで実行したビルドの情報やビルド・テストの失敗報告、コードカバレッジ解析、そしてメモリ使用状況などがここに集まる。ビルドのスケジュールをたてたり実行したりするのはリモートクライアント側で、ビルドレポートは XML 形式で送信する。ビルド結果の送信は "公式の" ビルドクライアントで行うこともできるし、コアデベロッパー以外の開発者やユーザーが公開したビルドプロセスを自分のマシンで実行することもできる。

このシンプルなモデルが可能が実現できた理由は、CDash とその他の Kitware ビルド基盤の要素 (ビルド設定システムである CMake、テストランナーである CTest、そしてパッケージングシステムである CPack) が概念的に密結合していたことである。このソフトウェアが提供する仕組みを使うと、ビルドやテストそしてパッケージングのレシピを高度に抽象化して実装できる。その際に OS を意識する必要はない。

CDash のクライアント主導のプロセスは、クライアント側の CI プロセスを多くの面で単純化する。ビルドを実行するかどうかを決めるのはクライアント側なので、クライアント側の状況 (時刻や負荷など) を考慮にいれてビルドを始めることができる。望みに応じてクライアントを増やしたり減らしたりするしてビルドを手伝ったり、ビルドを "クラウドで" 行うこともできる。ビルドの成果物を中央サーバーに送るのも、単純なアップロードで済む話だ。

しかし、このレポーティングモデルを採用した代償として、CDash には Buildbot が持つ多くの便利な機能が欠けている。中央管理型のリソース制御機能はないし、事前に登録済みでないクライアントを分散環境に投入することもできない。進捗レポート機能も実装されていない。実装するにはビルド状況のインクリメンタルな更新をサーバーが許可しないといけな

い。そしてもちろん、全体にビルド要求を出したり、チェックインに反応して匿名クライアントにビルドさせることもできない—クライアントはすべて信頼できないものとみなさなければならない。

最近、CDash に新機能が追加されてクラウドビルドシステム “@Home” が使えるようになった。これは、クライアントが CDash サーバーに対してビルドサービスを提供する仕組みだ。クライアントがサーバーをポーリングしてビルドリクエストを受けとり、そのリクエストを処理して、結果をサーバーに返す。2010 年 10 月時点の実装では、ビルドのリクエストはサーバー側で手動で行う必要がある。そして、クライアントはサーバーに接続しないとサービスを提供できない。しかし、これを素直に拡張すればより汎用的なビルドモデルが作れる。つまり、サーバー側から自動的にビルドリクエストを送ったら、対応可能なクライアントが処理してくれるというモデルだ。“@Home” システムは、後で説明する Pony-Build システムと非常に似た概念である。

実装モデル**: Jenkins**

Jenkins は幅広く使われている継続的インテグレーションシステムで、Java で書かれている。2011 年初期までは Hudson という名前で知られていた。スタンドアロンの CI システムとしてローカルシステム上で動かすこともできるし、リモートビルドの調整役として使うこともできる。あるいは、リモートでのビルドの情報を受け取るだけの役割としても使うことができる。JUnit のユニットテストやコードカバレッジレポートで使われている標準の XML をうまく活用し、さまざまなテストツールからのレポートを統合する。Jenkins は元々Sun が作り始めたものだが、さまざまな場所で使われており、しっかりとしたオープンソースコミュニティがついている。

Jenkins はハイブリッドモードで動作する。デフォルトはマスターサーバーでビルドを実行するが、さまざまなスタイルのリモートビルドも (サーバー側からでもクライアント側からでも) 実行できる。Buildbot と同様、本来は中央サーバーが管理するように作られている。しかし、さまざまな分散ジョブ実行機構に対応するようになった。仮想マシンの管理機能も含む。

Jenkins は複数のリモートマシンを管理することができる。接続はマスター側から SSH で確立することもできるし、クライアント側から JNLP (Java Web Start) で確立することもできる。この接続は双方向で、オブジェクトやデータもシリアル化してやりとりすることができる。

Jenkins にはしっかりとしたプラグイン機構が組み込まれており、この接続の詳細を抽象化している。そのおかげで、多くのサードパーティのプラグインがバイナリビルドや結果のデータを扱えるようになっている。

中央サーバーが管理するジョブ用として、Jenkins には “locks” プラグインが用意されている。このプラグインはジョブを並列実行させないようにするものだが、2011 年 1 月の時点では未完成だ。

実装モデル: Pony-Build

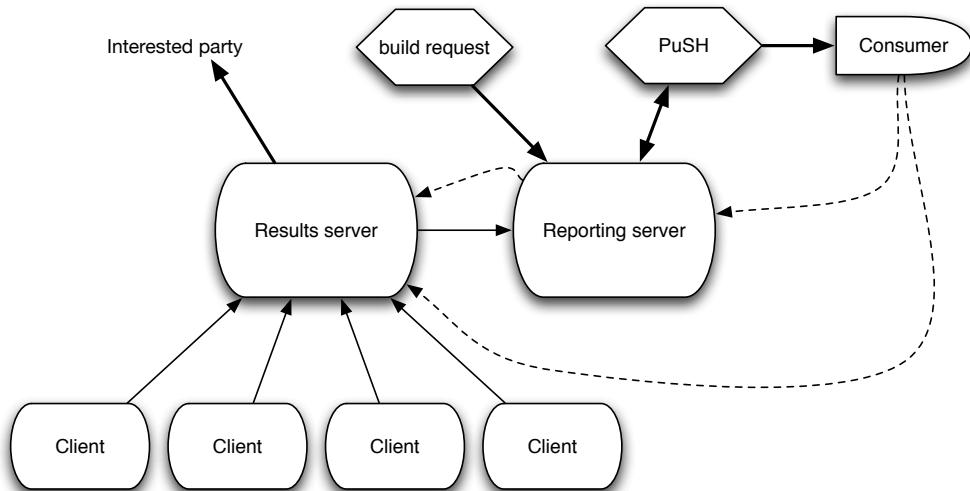



図 6.4: Pony-Build のアーキテクチャ

Pony-Build は、分散型の CI システムの概念実証モデルとして Python で作られた。図 6.4 に示す三つのコアコンポーネントで構成されている。結果サーバーが中央データベースとして働き、個々のクライアントから受け取ったビルド結果を保持する。クライアントはそれぞれ独立してすべての設定情報やビルドコンテキストを保持しており、軽量なクライアントライブラリで VCS のリポジトリにアクセスしたりビルドプロセスを管理したり、結果をサーバーに通信したりといったことができる。レポートサーバーは必須ではない。ここにはシンプルなウェブインターフェイスが組み込まれており、ビルド結果の報告や新しいビルドの要求を行う。我々の実装では、レポートサーバーと結果サーバーは単一のマルチスレッドプロセスで動作する。しかし API レベルでの結合は緩く、個別に動くようにも容易に変更できる。

基本モデルに加えてさまざまな WebHook や RPC 機構が用意されており、ビルドや変更通知そしてビルドに関する調査を支援する。たとえば、VCS のコードリポジトリの変更通知をビルドシステムと直接結び付けるのではなく、リモートからのビルドリクエストを直接レポートシステムに回し、レポートシステムがそれを結果サーバーに伝えるようにする。同様に、メールやインスタントメッセージングなどを使って新しいビルドを直接レポートサーバーにプッシュ通知するのではなく、通知の制御には PubSubHubbub (PuSH) を使っている。これにより、さまざまなアプリケーションが "興味のある" イベント (今のところは新しいビルドと失敗したビルドに限られる) の通知を PuSH WebHook で受け取れるようになっている。

このような疎結合のモデルを採用する利点は多い。

通信の容易性: 基本となるアーキテクチャ上のコンポーネントや WebHook のプロトコルは

極めて容易に実装でき、基本的なウェブプログラミングの知識さえあればよい。

変更の容易性: 新しい通知方式や新しいレポートサーバー用インターフェイスを実装するのは極めて容易である。

多言語のサポート: さまざまなコンポーネントがお互い WebHook で呼び合っている。大半のプログラミング言語は WebHook をサポートしているので、コンポーネントごとに別々の言語で実装することができる。

テストのしやすさ: 各コンポーネントは完全に分離していてモックがあるので、システムのテストを簡単に実行できる。

設定の容易性: クライアント側の要件は最小限で、Python そのもの以外に必要となるライブラリはひとつだけである。

サーバーの負荷の最小化: 中央サーバーにはクライアントを制御する責任が事実上ないと言えるので、隔離されたクライアントをサーバーと連携せずに並列に実行させることができる。報告時を除いて、サーバーに負荷をかけることはない。

VCS との統合: ビルドの設定は完全にクライアント側で行われるので、VCS に含めることもできる。

結果へのアクセスの容易性: ビルド結果を取得したいアプリケーションを書くためのプログラミング言語は、XML-RPC リクエストを扱えるものなら何でもよい。ビルドシステムのユーザーには結果サーバーやレポートサーバーへのネットワークレベルでのアクセス権限を与えることもできるし、レポートサーバーの独自インターフェイスを使ってもよい。ビルドクライアントに必要なアクセス権限は、結果サーバーへの結果の送信権だけである。

残念ながら、深刻な**弱点**も多い。これは CDash のモデルと同様である。

ビルド要求を送るのが難しい: この弱点の原因は、ビルドクライアントが結果サーバーから完全に独立しているということだ。クライアント側からサーバー側にビルド要求があるかどうかを確認することはできるかもしれない。しかしこれは負荷が高く、待ち時間も長くなる。それ以外の手段をとるなら、指揮・制御のための接続を確立してサーバー側からクライアント側にビルドリクエストを直接通知しなければならない。システムはさらに複雑になり、分散型ビルドクライアントの利点を損ねてしまう。

リソースロックのサポートが貧弱: リソースロックを管理するために RPC の仕組みを提供するのは簡単だが、クライアントポリシーを強制するのはずっと難しい。CDash のような CI システムはクライアント側を信頼することを前提としているが、クライアント側はそのつもりがなくても失敗してしまうかもしれない (ロックの解放を忘れるなど)。堅牢な分散型ロックシステムを実装するのは困難であり、余計な複雑さを持ち込んでしまう。たとえば、マスターリソースのロックを信頼できないクライアントに提供するには、ロックをつかんで離さないクライアントに対するポリシーをマスターロックコントローラー側で決めておく必要がある。これは、クライアントがクラッシュしたり、デッドロックが発生したりした場合の対策となる。

リアルタイムの監視処理が貧弱: リアルタイムでビルドを監視したりビルドプロセス自体を制御したりする仕組みを実装するのは、常に接続が持続しているシステムでないと困難である。Buildbot がクライアント主導のモデルに比べてはるかに優れている点のひとつは、時間のかかるビルドの途中経過を調べやすいというところだ。これは、ビルドの結果がマスター側のインターフェイスにインクリメンタルに送られてくるからである。さらに、Buildbot は制御用の接続を保持しているので、もし時間のかかるビルドが途中で (設定ミスや間違ったチェックインなどで) 失敗した場合は、そこでビルドを中止できる。そのような機能を Pony-Build(結果サーバーからクライアント側への連絡ができない) に追加するには、クライアント側から定期的にポーリングさせるかクライアント側への接続を確立するかのどちらかの仕組みが必要となる。

Pony-Build が提起する CI のその他ふたつの側面は、**レシピ**をいかに実装するかということ、そして**信頼**をどのように管理するのかということだ。これらはともに関連する問題である。というのも、レシピはビルドクライアント上の任意のコードを実行するからである。

ビルドレシピ

ビルドレシピは、使いやすいレベルの抽象化をしてくれるものだ。特に、クロスプラットフォームな言語で作られていたりマルチプラットフォームなビルドシステムを使っていたりする場合に役立つ。たとえば CDash は、非常に厳しいレシピに依存している。CDash を使うソフトウェアのほとんどすべては CMake や CTest そして CPack でビルドされており、これらのツールはマルチプラットフォームな課題に対応するよう作られている。これは、継続的インテグレーションシステムの視点で考えると理想的な状況だ。というのも、CI システムとしてはすべての課題をビルドツール群に丸投げするだけでよくなるからである。

しかし、これは必ずしもすべての言語やビルド環境で成り立つわけではない。Python の世界では distutils や distutils2 を使ってソフトウェアのビルドやパッケージングを行うことが標準になりつつある。しかし、テストを探して実行したりその結果を収集したりする仕組みにはまだ標準が確立されていない。さらに、より複雑な Python パッケージの多くは自前のビルドロジックをシステムに組み込んでおり、distutils の拡張機構 (任意のコードを実行できる) を通してそれを使っている。たいていのビルドツールで、同じような状況が見られる。標準的なコマンドが用意されていても、常に例外や拡張があるものだ。

ビルドやテストそしてパッケージングのレシピは、このようにいろいろ問題がある。なぜなら次のふたつの問題を解決しなければならないからだ。まず、プラットフォーム非依存な形式で指定しないといけない。単一のレシピを複数のシステム上のソフトウェアのビルドに使えるようにするためだ。そして、ビルドされるソフトウェアにあわせてカスタマイズ可能でなければならない。

信頼

さらに、第三の問題がある。CI システムのレシピを幅広く使い始めると、信頼しなければならない外部のシステムが増えてしまう。ソフトウェア自身が信頼できる (CI クライアントは任意のコードを実行できるので) だけでは不十分で、さらにレシピも信頼できなければならない (レシピもまた任意のコードを実行できるので)。

こういった信頼に関する問題は、しっかりと制御された環境なら扱いやすい。たとえば、ビルドクライアントや CI システムを内部プロセスの一環として扱うような企業がそれにあたる。しかし、その他の環境では、サードパーティがビルドサービスを提供することもある。たとえばオープンソースプロジェクト向けなどだ。理想的な解決策は、標準のビルドレシピをコミュニティレベルでソフトウェアに含められるようにすることだ。Python コミュニティは、これを distutils2 を使って行っている。もうひとつの解決策は、たとえばデジタル署名したレシピを使うことだ。そうすれば、信頼できる個人が書いたレシピを署名付きで配布でき、CI クライアントはそのレシピが信頼に値するかどうかを調べることができる。

どのモデルを選ぶか

経験上、疎結合な RPC や Webhook コールバックベースのモデルによる継続的インテグレーションは非常に実装しやすいものだ。複雑な結合にからむ密接な連携に関する要件を一切無視しさえすれば。基本的なリモートチェックアウトやビルドの実行には、ビルドをローカルで行うかリモートで行うかにかかわらず共通した設計上の制約がある。ビルドに関する情報 (成功/失敗など) の収集は、基本的にはクライアント側からの要求に基づいて行う。アーキテクチャによる情報の追跡も結果による情報の追跡も、同じ基本要件を必要とする。したがって、基本的な CI システムはレポーティングモデルを使えば極めて容易に実装できる。

疎結合のモデルも、非常に柔軟で拡張性のあるものであることがわかった。新たな結果報告の仕組みや通知の仕組み、あるいはビルドレシピを追加したりするのが容易になる。各コンポーネントが明確に分かれており、きちんと独立しているからである。分割されたコンポーネントはそれぞれやるべき作業が明確になっており、テストもしやすければ変更もしやすくなる。

CDash のような疎結合のモデルでリモートビルドをするときの唯一の困難は、ビルドの協調である。ビルドの開始や停止、進行中のビルドの状況報告、そして各クライアント間でのリソースロックの調整などが技術的に要求される。これらは、それ以外のモデルの実装ではあまり問われないものだ。

これらを踏まえて得られる結論は、疎結合のモデルのほうが全般的に “より優れている” ということだ。しかしそう言えるのは、ビルドの協調が不要な場合のみである。ビルドの協調が必要かどうかは、CI システムを利用するプロジェクトによって決まる。

6.3 将来

Pony-Build について検討しているときに、将来の継続的インテグレーションシステムにあったらいいなという機能をいくつか思いついた。

言語不可知なビルドレシピ群: 現状の継続的インテグレーションシステムはどれも、車輪の再発明をして独自のビルド設定言語を用意している。これは明らかにばかげた話だ。一般的に使われているビルドシステムは十種類にも満たないだろうし、テストランナーだっておそらく数十種類程度だろう。にもかかわらず、どの CI システムも新しい独自の方法でビルドを指定したり実行するテストコマンドを指定したりする。実際のところ、よく似た CI システムが氾濫している理由の一つがこれではないだろうか。それぞれのプログラミング言語やコミュニティが構成管理の仕組みを実装し、自分たちの使いなれたビルドシステムやテストシステムに合わせて調整し、同じような機能を持ったシステム上にそのレイヤーをかぶせているのだ。何かドメイン特化言語 (DSL) を作り、高々数十種類しかないビルドツールやテストツールで使われるオプションを表せるようにしておけば、長い目で見たときに CI の世界を簡素化できるのではないだろうか。

ビルドやテストの報告用の共通フォーマット: ビルドシステムやテストシステムが、どんな情報をどんなフォーマットで提供すればいいのかに関する決まりが一切ない。もし何らかの共通フォーマットや標準規格ができれば、継続的インテグレーションシステムがビルドの詳細や概要をより簡単に提供できるようになるだろう。今のところそれに近い位置にあるのが、Perl コミュニティで使われている TAP(The Test Anywhere Protocol) や Java コミュニティで使われている JUnit の XML 出力形式である。これらは、実行したテストの数や成功と失敗の数、そしてファイル単位でのコードカバレッジの詳細といった情報を表すことができる。

レポーティングにおける粒度や内部調査機能の向上: あるいは、さまざまなビルドプラットフォームがよくできたフックシステムを提供し、その構成やコンパイルそしてテストのシステムをフックできるようになっていると便利だろう。このフックシステムが (共通のフォーマット以上の)API を提供してくれれば、CI システムはそれを使ってより詳細なビルド情報を取り出せる。

まとめ

これまでに解説してきた継続的インテグレーションシステムは、それぞれのアーキテクチャにうまくあてはまる機能を実装してきた。一方、ハイブリッドである Jenkins は、最初はマスター/スレーブモデルだったが、より疎結合なレポーティングアーキテクチャの機能をそこに追加した。

選んだアーキテクチャによってその機能が決まる、という結論にしたいところだが、もちろんそれはナンセンスだ。それよりは、選んだアーキテクチャがその後の開発の方向性に影

響を与え、特定の機能群を実装する流れになるのではないだろうか。Pony-Build を作っていた我々も驚いた。最初に選んだ CDash 形式のレポーティングアーキテクチャが、後の設計や実装の決断に大きく影響したのだ。実装上の選択の中には、Pony-Build で中央管理型の構成やスケジューリングシステムを回避した例のように実際の使用例に基づいたものもある。我々が必要としていたのはリモートビルドクライアントを動的に追加できることで、これは Buildbot ではサポートしづらいものだった。それ以外に Pony-Build で実装しなかった機能には、進捗レポートや中央管理型のリソースロックなどがある。これらも実装したかったが、どうしてもという希望もなしに追加するには少し複雑すぎた。

同じような理屈が、おそらく Buildbot や CDash そして Jenkins にもあてはまるだろう。どのツールにも、有用なのに実装されていないという機能がある。おそらくそれはアーキテクチャ上の非互換性が原因だと思われる。しかし、Buildbot や CDash のコミュニティのメンバーとの議論や Jenkins のウェブサイトの記述によると、これらはまず欲しい機能を選ぶところから始めて、それからその機能を実装しやすいアーキテクチャでの開発を進めたらしい。たとえば、CDash のコミュニティには比較的小規模なコア開発者チームがおり、中央管理型のモデルでソフトウェアを開発している。最優先で考えるのはソフトウェアをコアマシン群で動作させ続けることで、その次は技術力のあるユーザーからのバグ報告を受け付けることだ。一方 Buildbot が勢力を伸ばしているのは、複雑なビルド環境に多数のクライアントが絡み、共有リソースへのアクセスの調整を要するようなところである。Buildbot には柔軟な設定ファイルフォーマットがあり、スケジューリングや変更通知そしてリソースロックなどのさまざまなオプションを設定できる。そのあたりが好まれているのだろう。Jenkins が目指しているのは、使いやすさとシンプルな継続的インテグレーションだろう。設定用の GUI とローカルサーバーで動かすためのオプションもその一環だ。

オープンソース開発の社会学も、アーキテクチャと機能の相関関係における交絡因子となる。考えてみよう。もし、開発者たちがオープンソースプロジェクトを選ぶときの基準が、そのプロジェクトのアーキテクチャや機能が自分の用途に一致しているかどうかだとしたら? もしそうならば、彼らの貢献が反映されて、そのプロジェクトは今の機能をより伸ばすようになっていくだろう。その結果、プロジェクトは特定の機能セットにロックインされてしまうことになる。もともとその機能を好んで自ら選んだ人たちが開発を手伝うので、彼らが望む機能にうまくマッチしないアーキテクチャを避けるだろう。我々が Buildbot の開発に参加せず新たに Pony-Build を実装する道を選んだのも、まさにこの理由によるものだ。Buildbot のアーキテクチャは、何百何千ものパッケージをビルドするのには適さなかったのだ。

既存の継続的インテグレーションシステムは一般に、本質的に異なる二つのアーキテクチャのいずれかで構築されており、要求される機能のサブセットしか実装していない。CI システムが成熟してユーザー数が増えていくにつれて、さらに機能が追加されていくだろうと期待している。しかし機能を実装するには、選択したアーキテクチャによる制約があるかもしれない。今後どのように発展していくかが楽しみだ。

謝辞

グレッグ・ウィルソンやブレット・キャノン、エリック・ホルシャー、ジェシー・ノラー、ヴィクトリア・レイドラーらとは CI システム全般 (特に Pony-Build) について興味深い議論をさせていただいた。Pony-Build の開発には、ジャック・カールソンやファティマ・シェルカウイ、マックス・ライト、クシュブ・シャキャらの学生たちが協力してくれた。

第7章

Eclipse

Kim Moir

モジュール形式でソフトウェアを設計することは、周知の通り、とても難しいことだ。さまざまなコミュニティが書いた巨大なコードベース間での相互運用性、これもまた非常に管理しづらいものである。Eclipse は、これら両方をなんとかうまく成し遂げた。2010 年 6 月、Eclipse Foundation は Helios リリースを公開した。39 のプロジェクト、そして 40 以上の企業に属する 490 人のコミッター達が協力し、その基盤となるプラットフォームの機能を作り上げた。Eclipse が誕生した当初は、どんなアーキテクチャを思い描いていたのだろう？そして、それがどのように進化したのだろう？あるアプリケーションのアーキテクチャが、コミュニティの成長を促すということがあるのだろうか？そもそものはじまりを振り返ってみよう。

2001 年 11 月 7 日、あるオープンソースプロジェクトのバージョン 1.0 がリリースされた。その名は Eclipse。当時、Eclipse は「特定の何か専用ではなく、何にでも使える統合開発環境」と自称していた。このように一般的な説明にしたのには意図があった。アーキテクチャ上の狙いとして、単なるツール群ではなくフレームワークを作ることを考えていたのだ。モジュール形式で、スケーラブルなフレームワークだ。Eclipse はコンポーネントベースのプラットフォームを提供し、開発者用のツールを作る基盤としてそれを使うこともできた。拡張可能なアーキテクチャのおかげで、コアプラットフォームをベースにして当初の狙いを越えて拡張するコミュニティを生み出した。Eclipse はプラットフォームとして誕生したもので、Eclipse SDK はその概念を実証するプロダクトだった。Eclipse SDK はそれ単体でも使えるし、Eclipse SDK を使って Eclipse の新たなバージョンを作ることもできた。

オープンソースの開発者と聞いて真っ先に思い浮かべるありがちなイメージは、「他の人の役に立てたらいいな」と思いながら毎晩遅くまでバグ対応やイケてる新機能の追加をしているような人たち。誰に言われるでもなく、個人的な興味を満たすためだけに作業に取り組むような人たちだ。でも、Eclipse プロジェクトを振り返ってみると、そんな人たちばかりではなかった。最初期に寄贈されたコードの一部には、IBM が開発する VisualAge for Java をベースにしたものもあったのだ。Eclipse プロジェクトの最初のコミッターとなったのは、IBM の子会社である Object Technology International (OTI) の社員たちだった。彼らは同社の業務と

してフルタイムで Eclipse プロジェクトに取り組み、ニュースグループでの質問に答えたりバグ修正をしたり、新機能を実装したりしていた。その後、それに関心を示したいくつかのソフトウェアベンダーとの間で提携し、プロジェクトは拡大していった。Eclipse コンソーシアムに参加した最初のメンバーは、Borland、IBM、Merant、QNX Software Systems、Rational Software、RedHat、SuSE、そして TogetherSoft だった。

このプロジェクトに出資することで、参加各社はさまざまなノウハウを得て、Eclipse をベースにした商用製品を出荷することもできるようになった。参加各社の判断で社員にオープンソースソフトウェアの改良をさせて、それをベースにした自社製品に役立てるという意味で、これは Linux カーネルへの出資に似ている。2004 年初頭に Eclipse Foundation が設立され、日々拡大する Eclipse コミュニティの運営と拡大を担当することになった。この非営利組織は法人会員の会費で運営され、理事会が統治していた。現在では Eclipse コミュニティの多様性はさらに広がり、170 社以上の企業と 1000 人近くのコミッターを抱えるようになっている。

出だしたころは “Eclipse” と言えば SDK のことだったのだが、今ではもっと広い意味で知られるようになってきた。2010 年 7 月の時点で、eclipse.org の配下で開発されているプロジェクトは 250 にも及ぶ。C/C++や PHP そして Web サービスの開発を支援するツールもあれば、モデル駆動開発ツールやビルドツールなどもある。これらの各プロジェクトは、何らかのトップレベルプロジェクト (TLP) に属し、プロジェクト管理委員会 (PMC) が管理している。PMC はプロジェクトの上級メンバーで構成されており、技術的な方向性やリリース目標などを定めるために選出される。簡潔にまとめるために、本章で取り上げる範囲は Eclipse[1] の中の Eclipse SDK と、ランタイムの Equinox[2] プロジェクトのアーキテクチャに限定する。Eclipse には長い歴史があるので、まずは初期の Eclipse について説明してから 3.0、3.4、そして 4.0 についても取り上げる。

7.1 初期の Eclipse

21 世紀が始まったころ、世の中にはソフトウェア開発用のツールは山ほど存在したが、組み合わせて使えるものなどほとんどなかった。そんな中で Eclipse は、アプリケーション開発者に相互運用性をもたらすオープンソースプラットフォームを提供しようとした。そうすれば、開発者は新たなツールを書くことだけに専念できるようになる。ファイルシステムとのやりとりやソフトウェアのアップグレード機能、そしてソースコードリポジトリとの接続などといったインフラがらみの問題は気にせずに済むのだ。Eclipse で一番よく知られているのは、おそらく Java Development Tools (JDT) だろう。JDT の狙いは、お手本となる Java 開発ツール群を作ることだった。他の言語用のツールを作ろうとする人たちが、これを参考にできるようにする意図があった。

[1] http://www.eclipse.org
[2] http://www.eclipse.org/equinox

Eclipse のアーキテクチャに深入りする前に、まずは開発者にとっての Eclipse SDK がどのようなものかを見ていこう。Eclipse を立ち上げてワークベンチを選ぶと、Java パースペクティブが表示される。パースペクティブとは、現在使用中のツール専用にビューやエディターの配置を管理する仕組みだ。

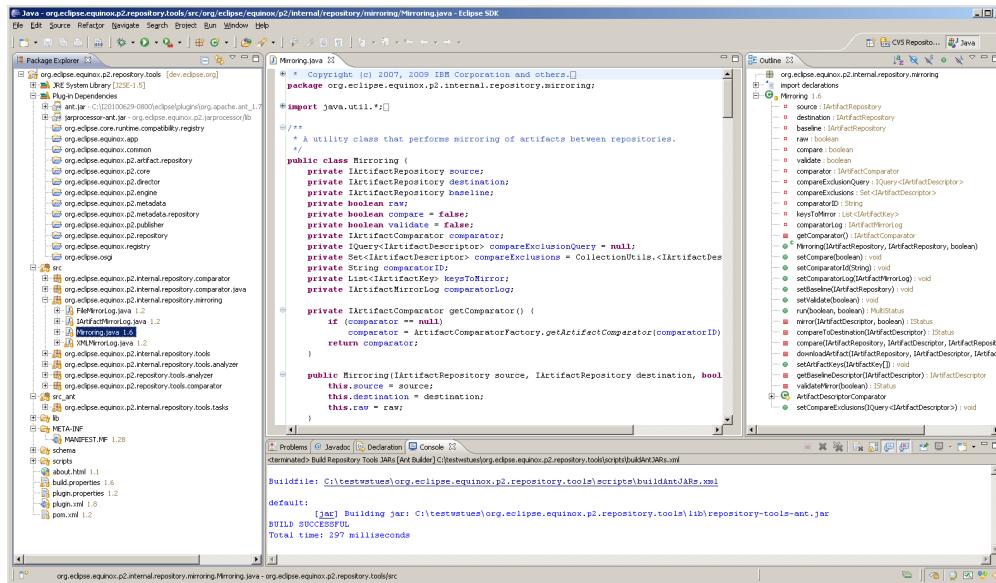



図 7.1: Java パースペクティブ

初期バージョンの Eclipse SDK のアーキテクチャには三つの主要な要素があり、それぞれが三つの主要なサブプロジェクトに対応していた。プラットフォーム、JDT (Java Development Tools)、そして PDE (Plug-in Development Environment) だ。

プラットフォーム

Eclipse プラットフォームは Java で書かれていて、実行するには Java VM が必要だ。プラグインと呼ばれる、機能ごとの小さな単位で構成されている。プラグインが、Eclipse のコンポーネントモデルの基盤となる。プラグインは基本的に JAR ファイルであり、その説明や依存関係、使いかた、そして拡張方法などがマニフェストに書かれている。このマニフェスト情報は、当初はプラグインのルートディレクトリにある `plug-in.xml` というファイルに格納されていた。JDT は、Java での開発用のプラグインを提供する。Plug-in Development Environment (PDE) は、プラグインを開発して Eclipse を拡張するためのツールを提供する。Eclipse のプラグインは Java で書かれているが、Java のコードを書く以外にも貢献の余地はある。オンラインドキュメントの HTML を書いたりなどといったものだ。プラグインごとに、自身のクラスローダーを持っている。あるプラグインと別のプラグインとの依存関係を表す

には、plugin.xml の requires ステートメントを利用する。org.eclipse.ui プラグインのplugin.xml を見てみよう。プラグインの名前やバージョン、さらに、別のプラグインからインポートしないといけない依存関係も記されている。

```
<?xml version="1.0" encoding="UTF-8"?>
<plugin
   id="org.eclipse.ui"
   name="%Plugin.name"
   version="2.1.1"
   provider-name="%Plugin.providerName"
   class="org.eclipse.ui.internal.UIPlugin">

   <runtime>
      <library name="ui.jar">
         <export name="*"/>
         <packages prefixes="org.eclipse.ui"/>
      </library>
   </runtime>
   <requires>
      <import plugin="org.apache.xerces"/>
      <import plugin="org.eclipse.core.resources"/>
      <import plugin="org.eclipse.update.core"/>
      :       :        :
      <import plugin="org.eclipse.text" export="true"/>
      <import plugin="org.eclipse.ui.workbench.texteditor" export="true"/>
      <import plugin="org.eclipse.ui.editors" export="true"/>
   </requires>
</plugin>
```

Eclipse プラットフォーム上での開発にかかわる人を増やすには、プラットフォームに貢献できるような仕組みが必要だ。そして、その貢献を受け入れる体勢も必要になる。それを実現したのが拡張と拡張ポイントで、これも Eclipse コンポーネントモデルの要素の一つだ。エクスポート機能で、他のプラグインが自身の拡張を書くときのインターフェイスを指定する。これが、プラグインの外部から使えるクラスを制限する。また、プラグインの外部から使えるリソースを制限する機能も提供する。すべての public メソッドやクラスが使えるのとは対象的だ。エクスポートされたプラグインは、公開 API だとみなされる。それ以外はすべて、内部的な実装の詳細だ。たとえば Eclipse のツールバーにメニュー項目を追加するようなプラグインを書きたければ、org.eclipse.ui プラグインの拡張ポイント actionSets を利用する。

```
<extension-point id="actionSets" name="%ExtPoint.actionSets"
                 schema="schema/actionSets.exsd"/>
<extension-point id="commands" name="%ExtPoint.commands"
                 schema="schema/commands.exsd"/>
<extension-point id="contexts" name="%ExtPoint.contexts"
                 schema="schema/contexts.exsd"/>
<extension-point id="decorators" name="%ExtPoint.decorators"
                 schema="schema/decorators.exsd"/>
<extension-point id="dropActions" name="%ExtPoint.dropActions"
                 schema="schema/dropActions.exsd"/> =
```

拡張ポイント org.eclipse.ui.actionSet を拡張してメニュー項目を追加するコードは、このようになる。

```
<?xml version="1.0" encoding="UTF-8"?>
<plugin
   id="com.example.helloworld"
   name="com.example.helloworld"
   version="1.0.0">
   <runtime>
      <library name="helloworld.jar"/>
   </runtime>
   <requires>
      <import plugin="org.eclipse.ui"/>
   </requires>
   <extension
         point="org.eclipse.ui.actionSets">
      <actionSet
            label="Example Action Set"
            visible="true"
            id="org.eclipse.helloworld.actionSet">
         <menu
               label="Example &Menu"
               id="exampleMenu">
            <separator
                  name="exampleGroup">
            </separator>
         </menu>
         <action
               label="&Example Action"
               icon="icons/example.gif"
               tooltip="Hello, Eclipse world"
               class="com.example.helloworld.actions.ExampleAction"
               menubarPath="exampleMenu/exampleGroup"
               toolbarPath="exampleGroup"
               id="org.eclipse.helloworld.actions.ExampleAction">
         </action>
      </actionSet>
   </extension>
</plugin>
```

Eclipse を立ち上げるときに、ランタイムプラットフォームはインストールされたプラグインのマニフェストを走査し、プラグインレジストリを構築してメモリに展開する。拡張ポイントとそれに対応する拡張は、名前でマッピングされる。できあがったプラグインレジストリは、Eclipse プラットフォームが提供する API で参照できるようになる。レジストリはディスクにキャッシュされ、Eclipse を再起動したときにもその情報を読み込める。すべてのプラグインは、起動時にレジストリに登録される。しかし、アクティブになる (クラスがロードされる) のは、実際にそのコードが使われるときになってからである。この手法を、遅延アクティベーションと呼ぶ。これで、新しいバンドルを追加したときのパフォーマンスへの影響が抑えられるようになる。プラグインに関連づけられたクラスは、実際にそれを使うとき

までロードされないからだ。たとえば org.eclipse.ui.actionSet 拡張ポイントを拡張するプラグインは、ユーザーがツールバーのメニュー項目を選択するまでロードされない。

図 7.2: Example メニュー

このメニュー項目を生成するコードは、次のようになる。

```
package com.example.helloworld.actions;

import org.eclipse.jface.action.IAction;
import org.eclipse.jface.viewers.ISelection;
import org.eclipse.ui.IWorkbenchWindow;
import org.eclipse.ui.IWorkbenchWindowActionDelegate;
import org.eclipse.jface.dialogs.MessageDialog;

public class ExampleAction implements IWorkbenchWindowActionDelegate {
    private IWorkbenchWindow window;

    public ExampleAction() {
    }

    public void run(IAction action) {
        MessageDialog.openInformation(
            window.getShell(),
            "org.eclipse.helloworld",
            "Hello, Eclipse architecture world");
    }

    public void selectionChanged(IAction action, ISelection selection) {
    }

    public void dispose() {
    }

    public void init(IWorkbenchWindow window) {
        this.window = window;
    }
}
```

ユーザーがツールバーの新しいメニュー項目を選択すると、拡張リポジトリを問い合わせてその拡張ポイントを実装するプラグインを探す。拡張を提供するプラグインがそのインスタンスを作成し、プラグインをロードする。プラグインがアクティブになると、ExampleAction のコンストラクタが走る。そして Workbench アクションデリゲートを初期化する。ワークベ

ンチ内での選択が変わってデリゲートが作られたら、アクションを変更できるようになる。ここでは、メッセージダイアログを出して “Hello, Eclipse architecture world” というメッセージを表示する。

このように拡張可能なアーキテクチャこそが、Eclipse を取り巻く環境が大きく成長した一つの要因だ。企業でも個人でもそれぞれ新しいプラグインを作れるし、作ったプラグインを商品として売ってもかまわない。

Eclipse で最も重要な概念は、**すべてはプラグインである**ということだ。Eclipse プラットフォームに含まれるプラグインもあれば自作のプラグインもある。いずれにせよ、プラグインはすべて、アプリケーションでのファーストクラスのコンポーネントとなる。図 7.3 は、初期の Eclipse でさまざまなプラグインが提供する機能の関連を示したものだ。

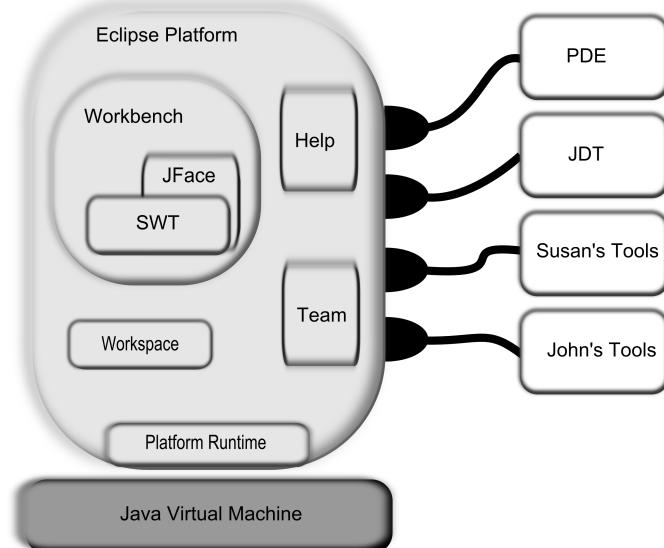



図 7.3: 初期の Eclipse のアーキテクチャ

ワークベンチは Eclipse プラットフォームのユーザーにとって最も身近な UI 要素で、Eclipse がユーザーのデスクトップ上でどのように表示されるのかを管理する。ワークベンチを構成するのは、パースペクティブとビューそしてエディタだ。エディタはファイルタイプと関連づけられており、何かのファイルを開いたときに適切なエディタが起動するようになっている。ビューの一例としては「問題」ビューがある。これは、Java のコードのエラーや警告を表示するものだ。これらのエディタとビューがパースペクティブを形成し、整頓された形式で各種ツールを表示する。

Eclipse ワークベンチは Standard Widget Toolkit (SWT) と JFace で作られている。SWT は、さらに詳しく見る価値があるだろう。ウィジェットツールキットには、大きく分けてネイティブ型とエミュレート型の二種類がある。ネイティブ型のウィジェットツールキットは、OS の

機能を直接呼んで、リストやボタンなどのユーザーインターフェイスコンポーネントを作る。そして、コンポーネントを使ったやりとりはOSが処理する。エミュレート型のウィジェットツールキットは、OSから離れてコンポーネントを実装する。マウスやキーボードの処理、描画、フォーカスなどの機能も、OS任せではなく自前で実装する。どちらの方式にも、それぞれ利点と欠点がある。

ネイティブウィジェットツールキットは「ピクセル単位で完璧」だ。ウィジェットの見た目は、他のデスクトップアプリと同じようになる。OSのベンダーは頻繁にウィジェットの見た目を変えるし、機能も追加する。ネイティブウィジェットツールキットなら、何もしなくてもその更新が適用される。残念ながら、ネイティブツールキットの実装は難しい。ベースになるOSレベルの実装はOSによって大きく異なるからだ。一貫性に欠けるし、作ったプログラムは移植性に欠けるものになる。

エミュレート型ウイジェットツールキットは、見た目を自前で提供することもあればOSのウィジェットに似せることもある。ネイティブツールキットと比べたときの大きな利点は、その柔軟性だ (ただ、最近のネイティブツールキットはかなりの柔軟性を持っている。たとえばWindows Presentation Framework (WPF) などがそうだ)。ウィジェットを実装するコードもツールキット自身で実装しており、OS組み込みのコードではないので、ウィジェットの描画や振る舞いを好きなようにできる。エミュレート型ウィジェットツールキットを使ったプログラムは、移植性が高まる。ただ、初期のエミュレート型ウィジェットツールキットは不評だった。速度が遅いしOSのエミュレートも貧弱だし、デスクトップ上では場違いな感じだった。特に、当時のSmalltalk-80で書いたプログラムは、Smalltalk-80で書いたことが一目瞭然だった。エミュレート型ウィジェットを使っていたからだ。ユーザーは、それを見れば「ああ、Smalltalkのプログラムだな」とわかるし、そのせいでSmalltalkのアプリケーションがなかなか受け入れられなかった。

CやC++のような他の言語とは異なり、Javaの初期バージョンにはネイティブウイジェットツールキットが付属していた。Abstract Window Toolkit (AWT) だ。AWTは機能に制限があってバグが多く、一貫性もないと、さんざんな言われようだった。Sunも他社も、AWTでの苦い経験があったので、可搬性があって高速に動くネイティブウィジェットツールキットは実現不可能だと思うようになった。そこで登場したのがSwing。これは完全な機能を持つエミュレート型ウィジェットツールキットだ。

1999年頃に、OTIがJavaを使ってVisualAge Micro Editionという製品を作った。VisualAge Micro Editionの最初のバージョンはSwingを使っていたが、OTIはSwingに満足していなかった。初期のSwingはバグが多かったし、タイミングやメモリの問題もあった。また、当時のハードウェアでは満足なパフォーマンスを得られなかった。OTIは、Smalltalk-80やその他のSmalltalk実装用のネイティブウィジェットツールキットの実装に成功し、Smalltalkが受け入れられるようになる道を開いた。この経験をいかして、SWTの最初のバージョンが作られた。VisualAge Micro EditionとSWTは成功を収め、Eclipseの開発が始まったときもごく自然な流れでSWTを使うことになった。SwingではなくSWTを使うとしたEclipseの判断については、Javaコミュニティで賛否両論が出た。中には陰謀論を語る人もいたが、Eclipse

は成功した。SWT を使ったことで、他の Java プログラムとの差別化ができたのだ。Eclipse は高パフォーマンスを誇り、ピクセル単位で完全であり、簡単に言えば「これが Java のプログラムだとは信じられない」ようなものだった。

初期の Eclipse SDK は、Linux と Windows にだけ対応していた。2010 年になって、10 をこえる数多くのプラットフォームをサポートするようになった。どれか一つのプラットフォームでアプリケーションを書けば、それを複数のプラットフォームで公開できるのだ。Java 用に新たなウィジェットツールキットを書くというのは、当時の Java コミュニティでは議論を引き起こしがちな問題だった。しかし Eclipse のコミッターは、それを踏まえた上で、最高のデスクトップ体験をもたらすためには新たなウィジェットツールキットを書く価値があると判断した。この主張は今でもあてはまり、何百万行ものコードが SWT 上に実装されている。

JFace は SWT の上位に位置するレイヤーで、一般的な UI プログラミングのタスクを担当する。たとえば、設定画面やウィザードなどのフレームワークだ。SWT と同様、JFace もさまざまなウィンドウシステムで動くように作られている。しかし、JFace はピュア Java で書かれており、ネイティブプラットフォームのコードは一切含まない。

Eclipse プラットフォームでは、統合されたヘルプシステムも提供している。これは、トピックと呼ばれる小さな情報単位をベースにしたシステムだ。一つのトピックは、ラベルとその場所への参照で構成されている。場所として指定できるのは、HTML ドキュメントファイルあるいはリンクを記した XML ドキュメントだ。トピックをひとまとめにして、目次 (Table of Contents: TOC) が作られる。トピックを葉だと考えると、TOC はその構造を表す枝と見なせる。ヘルプのコンテンツをアプリケーションに追加するには、拡張ポイント `org.eclipse.help.toc` を拡張する。ちょうど、次に示す `org.eclipse.platform.doc.isv` の `plugin.xml` のようになる。

```
<?xml version="1.0" encoding="UTF-8"?>
<?eclipse version="3.0"?>
<plugin>

<!-- ===================================================================== -->
<!-- Define primary TOC                                                    -->
<!-- ===================================================================== -->
   <extension
         point="org.eclipse.help.toc">
      <toc
            file="toc.xml"
            primary="true">
      </toc>
      <index path="index"/>
   </extension>
<!-- ===================================================================== -->
<!-- Define TOCs                                                           -->
<!-- ===================================================================== -->
   <extension
         point="org.eclipse.help.toc">
      <toc
```

```
            file="topics_Guide.xml">
      </toc>
      <toc
            file="topics_Reference.xml">
      </toc>
      <toc
            file="topics_Porting.xml">
      </toc>
      <toc
            file="topics_Questions.xml">
      </toc>
      <toc
            file="topics_Samples.xml">
      </toc>
   </extension>
```

Apache Lucene を使って、オンラインヘルプのコンテンツのインデキシングや検索を行う。初期の Eclipse では、オンラインヘルプを Tomcat で Web アプリケーションとして提供していた。さらに、Eclipse 自身でヘルプを提供するために、ヘルププラグインのサブセットを使ってスタンドアロンのヘルプサーバーを使うこともできる。[3]

Eclipse には、チームでの開発をサポートする機能もある。ソースコードリポジトリとのやりとりやパッチの作成など、一般的なタスクを実現できる。ワークスペースはファイルとメタデータのコレクションを提供し、作業内容をファイルシステム上に保存できる。また、デバッガもあり、Java コードの問題を追跡できるし、言語固有のデバッガを作るためのフレームワークもある。

Eclipse プロジェクトのゴールの一つは、オープンソースの開発者らが必要に応じてプラットフォームを拡張できるようにすることだ。そのための方法の一つが、安定した API を提供することだった。API とは一種の技術的な契約で、アプリケーションの振る舞いを定義するものでもある。あるいは、一種の社会契約であると捕らえてもいい。Eclipse プロジェクトの信念は「API よ永遠なれ」だ。したがって、API を書くときには細心の注意を払って検討する。一度定めた API は半永久的に使われることになるからだ。安定した API は、クライアントと API 提供者との間の契約になる。この契約によって、クライアントは Eclipse プラットフォームに依存して API を長期間提供できるようになる。クライアント側では、辛いリファクタリングの必要はなくなる。よい API というものは、実装を改良していける程度の柔軟性も持っている。

Java Development Tools (JDT)

JDT が提供するのは、Java 用のエディタやウィザード、リファクタリング機能、デバッガ、コンパイラ、そしてインクリメンタルビルダーだ。コンパイラは、コンテンツ・アシストやナビゲーションなどの編集機能にも用いられる。Java SDK は Eclipse に同梱されていないの

[3] その一例が http://help.eclipse.org だ。

で、どの SDK を使うかはユーザーしだいということになる。いったいなぜ、JDT チームはわざわざコンパイラを書いて Eclipse の中で Java コードをコンパイルできるようにしたのだろう？最初期のコンパイラのコードは VisualAge Micro Edition から寄贈されたものだった。ゆくゆくはこのコンパイラ上に各種ビルドツールを用意するつもりだったので、コンパイラを自前で書くのも極めて論理的な決断だった。この手法をとったおかげで、JDT ではコンパイラを拡張するための拡張ポイントを用意できるようになった。もしサードパーティのコマンドラインアプリケーションをコンパイラとして使っていたら、とてもそんなことはできなかっただろう。

自前のコンパイラを書いたことで、IDE 内でのインクリメンタル・ビルドに対応できるようになった。インクリメンタル・ビルダーが高いパフォーマンスを誇るのは、変更されたファイルおよびそれに依存するファイルだけを再コンパイルしているからだ。インクリメンタル・ビルダーはどのように動いているのだろう？Eclipse で Java のプロジェクトを作るときは、ワークスペースの中にリソースを作ってそこにファイルを格納することになる。Eclipse のビルダーは、ワークスペース内の`.java`ファイルを入力として受け取り、出力 (`.class`ファイル) を作る。ビルドステートを使って、そのファイルのワークスペース内での形式 (クラスかインターフェイスか) を知り、お互いどのように参照しあっているのかも知る。ビルドステートは、ソースファイルをコンパイルするたびにコンパイラがビルダーに渡す。インクリメンタル・ビルドを実行すると、ビルダーにはリソースの差分が渡される。この差分には、新しいファイルや変更されたファイル、削除されたファイルについての説明がある。ソースファイルが削除された場合は、それに対応するクラスファイルも削除される。新しいファイルや変更されたファイルは、コンパイルキューに追加される。キューの中のファイルが順番にコンパイルされ、前のクラスファイルと比較して、構造的な変更があったかどうかを調べる。構造的な変更とは、それを参照する別のクラスやインターフェイスに栄養を及ぼす変更のことだ。たとえばメソッドのシグネチャの変更や、メソッドの追加・削除がそれにあたる。構造的な変更があった場合は、それを参照するすべてのクラスやインターフェイスもキューに追加される。それらも変更があった場合は、コンパイル後のクラスファイルを出力フォルダに書き出す。そして、コンパイルされたものの参照情報を元にしてビルドステートを更新する。この流れを、キューが空になるまで繰り返す。コンパイルエラーが発生すると、Java エディタは問題マーカーを作る。長年にわたって、JDT が提供するツールは Java ランタイムのバージョンが新しくなるたびに大きく拡大してきた。

プラグイン開発環境 (PDE)

プラグイン開発環境 (Plug-in Development Environment: PDE) は、プラグインの開発やビルド、デプロイ、テストに使うツール群などを提供する。これらを使って、Eclipse の機能を拡張できる。Eclipse プラグインは Java の世界にないタイプの生成物だったので、ソースからプラグインに直接変換するビルドシステムがなかった。そこで PDE チームは PDE Build とい

うコンポーネントを作った。これは、プラグインの依存関係を調べてビルド用の Ant スクリプトを生成するものだ。

7.2 Eclipse 3.0: ランタイム、RCP そしてロボットたち

ランタイム

Eclipse 3.0 は、これまでの Eclipse のリリースの中でも最も重要なものの一つだろう。ひとつ前のリリースから、大きな変更が数多く加えられた。3.0 より前の Eclipse のアーキテクチャでは、Eclipse のコンポーネントモデルはプラグインで構成されていて、各プラグインが双方向にやりとりできるようになっていた。まず、プラグイン自身が依存関係を表す手段として、`plugin.xml` の中で `requires` ステートメントが使えた。プラグイン A がプラグイン B を require すると、プラグイン B が持つすべての Java クラスやリソースを (アクセス修飾子に応じて) プラグイン A から読めるようになる。各プラグインにはバージョンがあり、依存関係をバージョン付きで指定することもできた。次に、3.0 より前のコンポーネントモデルには**拡張**と**拡張ポイント**の仕組みがあった。歴史的に、Eclipse のコミッターはそれぞれ自前のランタイムを Eclipse SDK 用に書いて、クラスのロードや依存関係そして拡張や拡張ポイントを管理していた。

Equinox プロジェクトは、Eclipse のインキュベータープロジェクトとして誕生した。Equinox プロジェクトの目標は、Eclipse のコンポーネントモデルを既存の別のモデルに置き換えて、動的プラグインにも対応できるようにすることだった。当時検討されていたソリューションは、JMX や Jakarta Avalon そして OSGi などだ。JMX はその当時まだ発達途上のコンポーネントモデルだったので、候補から外れた。Jakarta Avalon の採用も見送った。プロジェクトが停滞しており、今後の発展が見込めなかったからだ。技術的な要件だけでなく、その技術をとりまくコミュニティも大事な判断材料だったのだ。Eclipse に特有の変更も組み込んでもらえるだろうか？活発に開発されていて、新たなメンバーを呼びこめているだろうか？Equinox チームは、コミュニティが最終的にどんな選択をしたかということを技術的要件と同様に重視していた。

その当時に使えそうだった各種コンポーネントモデルを調査して評価した結果、コミッターたちが選んだのが OSGi だ。なぜ OSGi だったのだろうか？まず、OSGi にはセマンティックなバージョン管理方式があって、それで依存関係を管理できた。また、モジュール方式に対応したフレームワークも用意されていた。これは JDK 自体には欠けている機能だった。パッケージを他のバンドルからも使えるようにするには明示的にエクスポートする必要があり、エクスポートしていないものは見えなくなる。OSGi は自前のクラスローダーを提供していたので、Equinox チームはこれまでの自作のローダーを維持し続ける必要がなくなった。Eclipse の世界以外でも幅広く使われているコンポーネントモデルで標準化したことで、より多くの人たちにアピールできるようになり、Eclipse を採用しやすくなった。

Equinox チームはこの選択に満足していた。既に OSGi には活発なコミュニティが存在したし、そのコミュニティと協力して、Eclipse で必要とする機能をコンポーネントモデルに組み込むこともできたからだ。たとえば、その当時の OSGi はパッケージレベルでしか要件をまとめられなかったが、Eclipse ではプラグインレベルで扱えるようにする必要があった。さらに、OSGi にはまだフラグメントの概念が組み込まれていなかった。Eclipse ではこの概念を使って、プラットフォームや環境に依存するコードを既存のプラグインに組み込んでいた。たとえば、Linux のファイルシステムでも Windows のファイルシステムでも使えるようなコードを書いたり、各国語への翻訳を扱ったりするときに、フラグメントを使っていたのだ。新しいランタイムとして OSGi を使うと決めたときに、コミッターたちがまず必要としたのは、オープンソースでのこのフレームワークの実装だった。評価の対象になったのは、Oscar(後の Apache Felix) や IBM の Service Management Framework (SMF) だった。当時の Oscar は研究プロジェクトで、公開範囲が限定されていた。最終的に採用したのは SMF だ。すでに他の製品での実績があることから、実用に耐えるであろうと判断した。Equinox の実装は、OSGi の仕様のリファレンス実装とされている。

互換性を確保するためのレイヤーも用意しているので、既存のプラグインも 3.0 の環境で動かせる。プラグインの開発者たちに「Eclipse 3.0 の新しい基盤に対応するためには修正が必要です」などと言ってしまえば、ツール開発用のプラットフォームとしての Eclipse の勢いをそいでしまう。Eclipse 上でツールを作っている人たちが期待するのは、単にプラットフォームがきちんと動き続けることだろう。

OSGi への切り替えに伴って、Eclipse のプラグインは「バンドル」と呼ばれるようになった。プラグインとバンドルはどちらも同じものだ。どちらも何らかの機能のサブセットを提供するモジュールで、自身についての説明はマニフェストの中のメタデータに保持する。以前のバージョンでは、依存関係やエクスポートするパッケージ、そして拡張と拡張ポイントは `plugin.xml` に書かれていた。OSGi バンドルでも、拡張や拡張ポイントは `plugin.xml` に記述する。というのも、これらは Eclipse 固有の概念だからである。残りの情報は `META-INF/MANIFEST.MF` に記述する。これは、OSGi におけるバンドルのマニフェストを書くファイルだ。この変更に対応するため、PDE は Eclipse に新たなマニフェストエディタを用意した。各バンドルには名前とバージョンがある。たとえば、`the org.eclipse.ui` バンドルのマニフェストは次のようになる。

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: %Plugin.name
Bundle-SymbolicName: org.eclipse.ui; singleton:=true
Bundle-Version: 3.3.0.qualifier
Bundle-ClassPath: .
Bundle-Activator: org.eclipse.ui.internal.UIPlugin
Bundle-Vendor: %Plugin.providerName
Bundle-Localization: plugin
Export-Package: org.eclipse.ui.internal;x-internal:=true
Require-Bundle: org.eclipse.core.runtime;bundle-version="[3.2.0,4.0.0)",
 org.eclipse.swt;bundle-version="[3.3.0,4.0.0)";visibility:=reexport,
```

```
 org.eclipse.jface;bundle-version="[3.3.0,4.0.0)";visibility:=reexport,
 org.eclipse.ui.workbench;bundle-version="[3.3.0,4.0.0)";visibility:=reexport,
 org.eclipse.core.expressions;bundle-version="[3.3.0,4.0.0)"
Eclipse-LazyStart: true
Bundle-RequiredExecutionEnvironment: CDC-1.0/Foundation-1.0, J2SE-1.3
```

Eclipse 3.1 からは、マニフェストでバンドルの必須実行環境 (BREE) を指定できるようになった。ここで指定するのは、そのバンドルを実行するために必要な最低限の Java 環境だ。Java コンパイラはバンドルや OSGi のマニフェストを理解しない。OSGi バンドルの開発用のツールは PDE が提供する。つまり、PDE がバンドルのマニフェストを読み取って、そのバンドル用の classpath を生成する。たとえば、マニフェストで実行環境として J2SE-1.4 を指定すると、ジェネリクスを使ったコードを書いたときにコンパイルエラーが表示されることになる。これで、自分の書いたコードがマニフェストの環境に従っていることを確認できる。

OSGi は、Java でモジュール方式を扱うためのフレームワークだ。OSGi フレームワークは、自己記述型のバンドルの集合を管理し、そのクラスローディングも扱う。個々のバンドルが自分用のクラスローダーを持っており、あるバンドル用の classpath を組み立てるために、マニフェストに書かれている依存関係を調べる。OSGi アプリケーションは、バンドルの集合となる。そのモジュール性をフルに活用するには、自作のバンドルの依存関係を、信頼できるフォーマットでユーザーに示せることが必要だ。そのため、マニフェストでは、そのバンドルが公開するパッケージを記述する。バンドルを使う側から見たときに、これらのパッケージが利用可能だということになる。このパッケージが、公開 API に対応する。API を利用するバンドル側は、対応するパッケージをインポートしておく必要がある。マニフェストで依存関係を記述するときには、バージョンの範囲も指定できる。先ほどのマニフェストの `Require-Bundle` を見ればわかるとおり、`org.eclipse.ui` が依存する `org.eclipse.core.runtime` バンドルのバージョンは 3.2.0 以上 4.0.0 未満であることが指定されている。

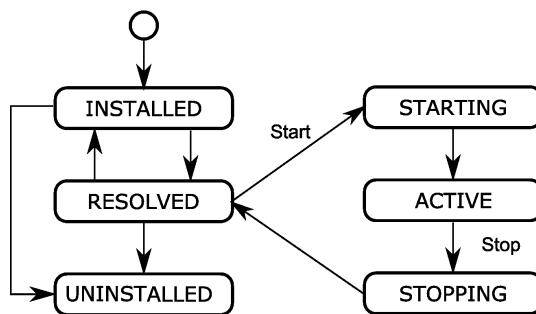



図 7.4: OSGi バンドルのライフサイクル

OSGi は動的なフレームワークで、バンドルのインストールや開始・停止、そしてアンインストールにも対応している。先述のとおり、遅延アクティベーションは Eclipse の大きな強みのひとつだ。プラグインのクラスは、実際に必要になるまでロードされないようになってい

た。OSGi バンドルも、この方式を使える。OSGi アプリケーションを立ち上げると、バンドルは「installed」の状態になる。依存関係がすべて満たされると、状態が「resolved」に変わる。この状態になれば、バンドル内のクラスをロードして実行できるようになる。「starting」は、アクティベーションポリシーにしたがってそのバンドルがアクティベートされた状態を意味する。いったんアクティベートすれば、バンドルは「active」な状態になる。この状態になれば、必要なリソースの確保や他のバンドルとのやりとりができるようになる。アクティベーターの stop メソッドを実行すると、アクティブなときに開いたリソースの後始末をしてから「stopping」状態に移行する。最終的にバンドルが「uninstalled」の状態になると、使えなくなる。

API は進化し続けるので、利用者側に API の変更を通知する仕組みも必要だ。一つの方法として考えられるのが、バンドルのバージョン名のつけかたを決めて、マニフェストでバージョンの範囲を指定できるようにするというものだ。OSGi では、図 7.5 に示すように四つの部分に分かれたバージョン体系を採用している。

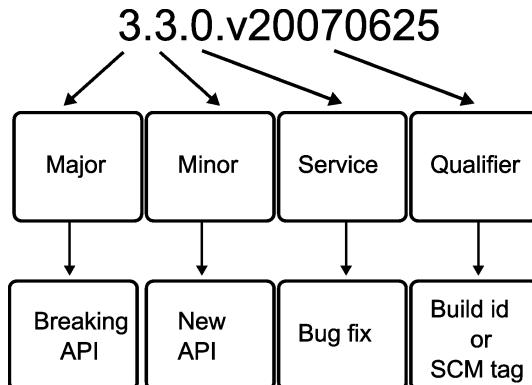



図 7.5: バージョン名のつけかた

OSGi のバージョン命名指針に従うと、個々のバンドルは、その名前と四つのバージョン番号からなる一意な識別子を持つことになる。id と version を指定すれば、あるバイト群を利用者側が特定できるようになる。Eclipse の慣習では、バンドルに何か変更を加えたときにはその変更の種類によってがージョンのどの部分を変えるかが決まる。もし API の互換生を崩してしまうような場合は、最初の部分 (major) を変更する。新しい API を追加するだけのようなときは、二番目の部分 (minor) を変更する。API には影響しないようなバグ修正の場合は、三番目の部分 (service) を変更する。四番目の部分は、ソース管理リポジトリのタグにあわせて変更し、そのビルドの ID を表す。

バンドル間の固定した依存関係を表す仕組み以外にも、OSGi にはサービスという仕組みが用意されている。これを使えば、バンドル間をより疎結合にできる。サービスとはいくつかのプロパティを持つオブジェクトで、OSGi サービスレジストリに登録されている。拡張については Eclipse の起動時にスキャンしたバンドルが拡張レジストリに登録されるが、サー

ビスの場合は動的に登録される。サービスを利用するバンドルは、そのサービスの規約を定義したパッケージをインポートする必要がある。フレームワークが、そのサービスの実装をサービスレジストリから取得する。

Java のクラスファイルには main メソッドがあるように、Eclipse にも起動用のアプリケーションが定義されている。Eclipse のアプリケーションは、拡張を使って定義する。たとえば Eclipse IDE 自身を起動するアプリケーションは `org.eclipse.ui.ide.workbench` で、これは `org.eclipse.ui.ide.application` バンドルの中で定義されている。

```
<plugin>
    <extension
         id="org.eclipse.ui.ide.workbench"
         point="org.eclipse.core.runtime.applications">
      <application>
         <run
               class="org.eclipse.ui.internal.ide.application.IDEApplication">
         </run>
      </application>
  </extension>
</plugin>
```

Eclipse はさまざまなアプリケーションを提供している。スタンドアロンのヘルプサーバーを起動するアプリケーションもあれば、Ant のタスクや JUnit のテストを実行するアプリケーションもある。

リッチクライアント・プラットフォーム (RCP)

オープンソースコミュニティで活動していて興味深いのは、自分たちがまったく予想しない使いかたでそのソフトウェアを使う人たちがいることだ。Eclipse のそもそもの狙いは、IDE を作ったり拡張したりするための基盤とツールを提供することだった。しかし、3.0 のリリースが近づいてきたころのバグレポートで明らかになったのだが、プラットフォームバンドルのサブセットを使ってリッチクライアント・プラットフォーム (RCP) アプリケーションを作ろうという動きが出始めた。多くの人は、これを Java アプリケーションとして認識していた。Eclipse は IDE に注力した作りになっていたので、バンドルの構造に少し手を入れて、そんな使い方にも対応できるようにした。ユーザーコミュニティに、より取り入れてもらいやすくするためだ。RCP アプリケーションは IDE の全機能を必要とするわけではない。そこで、いくつかのバンドルはさらに細かく分割して、RCP アプリケーションの構築にも使えるようにした。

RCP アプリケーションの実例としては、NASA のジェット推進研究所が開発したマーズ・ローバー・ロボットの監視に RCP を使っている例がある。さらに、バイオインフォマティックスのデータを可視化する Bioclipse やオランダの鉄道が列車の運行監視に使っている例もある。これらのアプリケーションに共通するのは、RCP が提供する機能を活用して自分たちは本来作りたいところに集中しようとチームが決断したことだ。彼らはその決断のおかげで時

間と資金を節約でき、自分たちのツールを作ることに注力できた。基盤となるプラットフォームの API は安定していて、長期間にわたるサポートが保証されている。

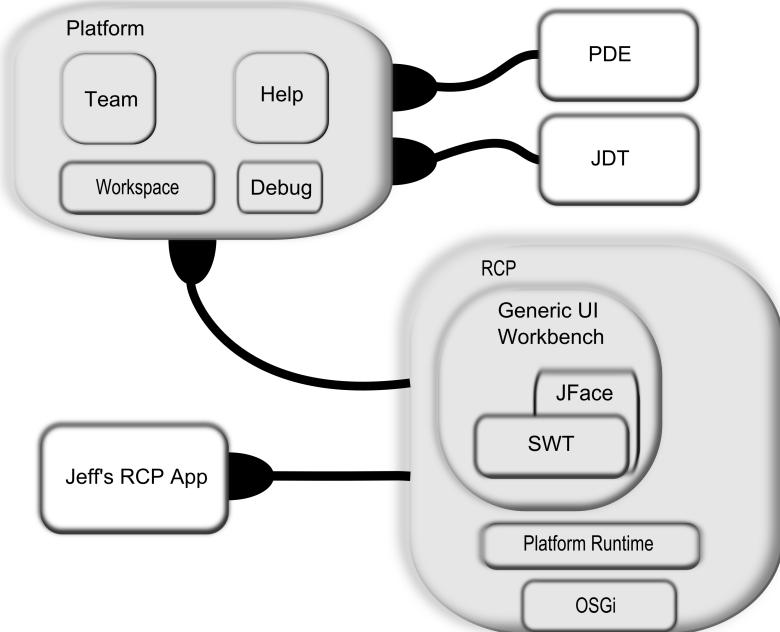



図 7.6: Eclipse 3.0 のアーキテクチャ

図 7.6 に示す 3.0 のアーキテクチャを見ると、Eclipse Runtime がまだ残っていてアプリケーションモデルと拡張レジストリを提供していることに気づくだろう。コンポーネント間の依存関係を管理するために、プラグインモデルは OSGi の管理下に置かれるようになった。これまでどおりに Eclipse を拡張して自分用の IDE を作ることもできるし、RCP アプリケーションフレームワークを使ってより汎用的なアプリケーションを作ることもできる。

7.3 Eclipse 3.4

アプリケーションを新しいバージョンに簡単に更新できて、新たな機能が使えるようになるというのは、あって当然の仕組みだと考えられている。たとえば Firefox なら、これはシームレスに行われる。Eclipse の場合はそんなに簡単にはいかなかった。更新マネージャーが導入されたことで、Eclipse に新たな機能を追加したり新しいバージョンに更新したりができるようになった。

アップデートやインストールの際に何が変わるのかを知るには、Eclipse の「フィーチャー」について理解する必要がある。フィーチャーとは PDE の生成物の一つで、複数のバンドルのセットを定義する。これらがひとまとまりになって、ビルドしたりインストールしたりでき

る形式になっている。フィーチャーの中に、別のフィーチャーを含めることもできる (図 7.7 を参照)。

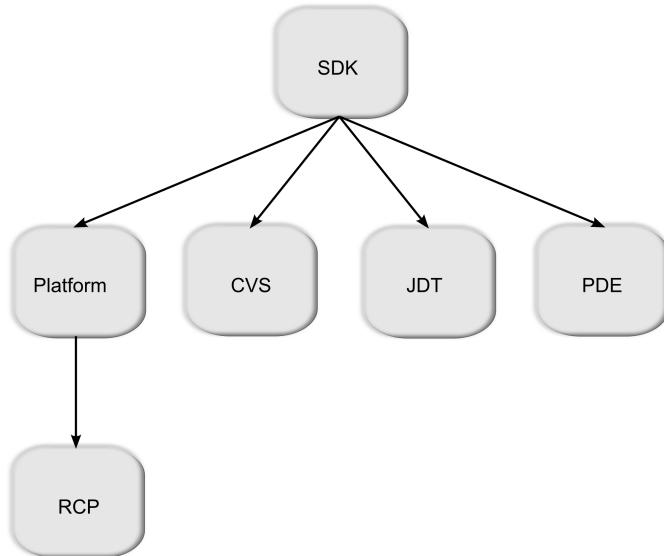



図 7.7: Eclipse 3.3 SDK のフィーチャーの階層

インストールされている Eclipse のうちの一つのバンドルが新しくなったので更新しようとした場合は、フィーチャー全体をアップデートする必要がある。フィーチャーはあまりきめ細かくない仕組みで、更新マネージャーが利用しているものだからである。たった一つのバンドルのためだけにフィーチャーを更新するのは非効率的だ。

PDE のウィザードを使って、フィーチャーの作成やワークスペース内でのビルドができる。feature.xml というファイルに、そのフィーチャーに含めるバンドルやバンドルのプロパティを定義する。フィーチャーには、バンドルと同様に名前やバージョンがある。フィーチャーの中に別のフィーチャーを含めることもできるし、含めるフィーチャーのバージョンを範囲で指定することもできる。フィーチャーに含めるバンドルは、そのプロパティも含めてリストで管理する。たとえば org.eclipse.launcher.gtk.linux.x86_64 フラグメントでは、OS(os) やウインドウシステム (ws)、そしてアーキテクチャ(arch) などを指定していることがわかるだろう。この場合、新しいリリースに更新するときには、このフラグメントは指定したプラットフォームにだけインストールされる。これらのプラットフォームに関するフィルタは、このバンドルの OSGi マニフェストに記述する。

```
<?xml version="1.0" encoding="UTF-8"?>
<feature
      id="org.eclipse.rcp"
      label="%featureName"
      version="3.7.0.qualifier"
      provider-name="%providerName"
```

```
      plugin="org.eclipse.rcp"
      image="eclipse_update_120.jpg">

   <description>
      %description
   </description>

   <copyright>
      %copyright
   </copyright>

   <license url="%licenseURL">
      %license
   </license>

   <plugin
         id="org.eclipse.equinox.launcher"
         download-size="0"
         install-size="0"
         version="0.0.0"
         unpack="false"/>

   <plugin
         id="org.eclipse.equinox.launcher.gtk.linux.x86_64"
         os="linux"
         ws="gtk"
         arch="x86_64"
         download-size="0"
         install-size="0"
         version="0.0.0"
         fragment="true"/>
```

Eclipse アプリケーションは、単にフィーチャーやバンドルの寄せ集めだけでできているわけではない。Eclipse 自身を立ち上げるためのプラットフォーム固有の実行ファイルもあればライセンスファイルもあるし、プラットフォーム固有のライブラリもある。これらについても Eclipse アプリケーションのファイル一覧に含まれる。

```
com.ibm.icu
org.eclipse.core.commands
org.eclipse.core.conttenttype
org.eclipse.core.databinding
org.eclipse.core.databinding.beans
org.eclipse.core.expressions
org.eclipse.core.jobs
org.eclipse.core.runtime
org.eclipse.core.runtime.compatibility.auth
org.eclipse.equinox.common
org.eclipse.equinox.launcher
org.eclipse.equinox.launcher.carbon.macosx
org.eclipse.equinox.launcher.gtk.linux.ppc
org.eclipse.equinox.launcher.gtk.linux.s390
```

```
org.eclipse.equinox.launcher.gtk.linux.s390x
org.eclipse.equinox.launcher.gtk.linux.x86
org.eclipse.equinox.launcher.gtk.linux.x86_64
```

これらのファイルは更新マネージャーでのアップデートができない。更新マネージャーはフィーチャーだけを扱うものだからだ。これらのファイルの多くはメジャーリリースのたびに更新される。つまり、新しいリリースがあるたびに zip を毎回ダウンロードしないといけないということだ。今インストールされているものを単純にアップデートするだけでは済まない。Eclipse のコミュニティでは、そんなことは到底受け入れられない。PDE は、プロダクトのファイルもサポートしていた。Eclipse RCP アプリケーションのビルドに必要なすべてのファイルを指定するものだ。しかし、更新マネージャーにはこれらのファイルをインストール環境に展開する仕組みがない。これは、ユーザーだけでなくプロダクトの開発者にとってももどかしいことだった。2008 年 3 月に、p2 が公開されて SDK に組み込まれた。これは、プロビジョニング用の新たな仕組みだ。過去との互換性を考慮して、更新マネージャーもまだ使えるようになっている。しかし、デフォルトでは p2 が有効になるように変わった。

p2 の概念

Equinox p2 は、要するに情報ユニット (IU) だ。IU とは、インストール仕様としている生成物の名前と ID に関する記述である。このメタデータには、生成物の機能 (何ができるのか) とその要件 (何に依存するのか) も書かれている。もし特定の環境でしか使えないのなら、それを表すフィルタも記述できる。たとえば、org.eclipse.swt.gtk.linux.x86 フラグメントが使えるのは Linux gtk x86 マシンにインストールしたときだけだ。本来、メタデータとはバンドルのマニフェストの情報を表したものである。生成物は、インストールされる単なるバイナリになる。関心の分離を実現するために、インストールされる内容とそのメタデータを完全に分離した。p2 のリポジトリには、メタデータのリポジトリと生成物のリポジトリの両方が含まれる。

プロファイルは、インストールされた IU の一覧だ。たとえば、Eclipse SDK は現在インストールされている内容を記したプロファイルを持っている。Eclipse の内部から、新しいバージョンへの更新をリクエストすることができる。そのときは、別の IU セットからなる新しいプロファイルを作る。プロファイルでは、そのインストール環境に関連するプロパティの一覧も提供する。たとえば OS やウィンドウシステム、そしてアーキテクチャなどのパラメータだ。また、インストール先のディレクトリや場所もプロファイルに格納する。プロファイルはプラファイル用のレジストリで管理され、ここには複数のプロファイルを格納できる。プロビジョニングを実行するのはディレクターの役割だ。ディレクターは、プランナーやエンジンとともに動作する。プランナーは既存のプロファイルを調べ、インストールされた環境を新しい状態に移行するために必要な操作を決める。エンジンは、実際のプロビジョニングを担当し、新しい生成物をディスクにインストールする。エンジンの一部として含まれているのがタッチポイントで、これは、インストールされたシステムのランタイム実装ととも

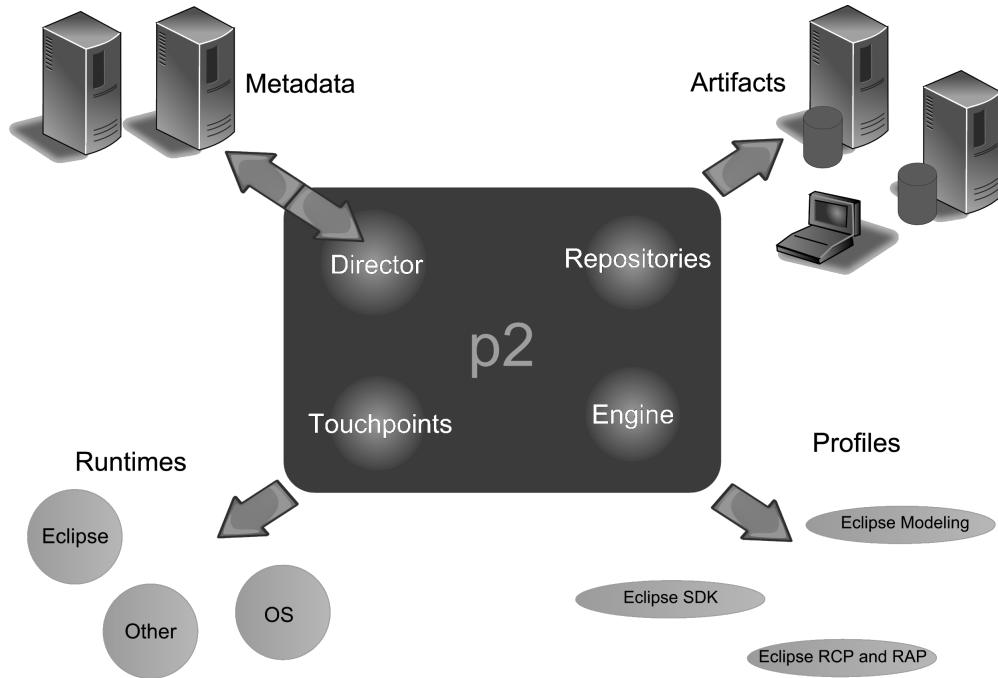



図 7.8: P2 の概念

に動作する。たとえば Eclipse SDK には Eclipse タッチポイントがあり、これがバンドルのインストール方法を知っている。Linux システムで Eclipse を RPM からインストールする場合、エンジンは RPM タッチポイントを使うことになる。また、p2 はインストールをインプロセスで行うこともできるし、ビルドなどのように別プロセスで行うこともできる。

新しいプロビジョニングシステムである p2 には、いろんなメリットがあった。Eclipse がインストールする生成物は、リリースのたびに更新される。前回のプロファイルがディスク上に残っているので、Eclipse を以前の状態に戻すこともできる。さらに、プロファイルとリポジトリがあれば、バグ報告をしてきたユーザーと同じインストール環境を再現できる。これで、バグ報告の内容を再現しやすくなる。p2 によるプロビジョニングは、単に Ecipse SDK の更新やインストールにだけ使えるというものではない。RCP や OSGi でも同様に使える。Equinox チームは、Eclipse Communication Framework (ECF) という別の Eclipse プロジェクトのメンバーと協力し、p2 リポジトリの生成物やメタデータを扱える方法を提供した。

p2 を SDK に組み込むにあたっては、Eclipse コミュニティ内でも活発な議論があった。更新マネージャーは Eclipse 自身のインストール環境をプロビジョニングするための必要最小限のソリューションでしかなかったので、ユーザーは、バンドルを導入するときには zip ファイルを展開してから再起動するという手順になれていた。この手法は、ベストエフォート型の仕組みでバンドルを解決するものだ。また、何らかのコンフリクトが発生したとしても、それを解決するのは実行時になる。インストール時ではない。制約があるのなら、実行時ではなく

てインストール時に解決しておくべきだ。しかし、ユーザーは、こんな問題があることに気づかないことが多い。バンドルがディスク上にあるのだから、動いてしかるべきだと思い込んでしまうのだ。以前のバージョンでは、Eclipse の提供する更新サイトは、単に JAR 化したバンドルやフィーチャーを並べたディレクトリ構造があるだけだった。シンプルな `site.xml` ファイルを用意して、そのサイトで提供するフィーチャーの名前を並べておけばよかった。p2 の時代になって、p2 リポジトリが提供するメタデータはもっと複雑なものになった。メタデータを作る方法は、ビルドプロセスに手を入れてビルド時にメタデータを生成させるか、既存のバンドルに対してジェネレータタスクを実行するかのいずれかになる。当初は、こういった変更の方法を説明したドキュメントが欠けていた。さらに、いつものことではあるが、新しいテクノロジーを一般公開すると、予期せぬバグが見つかって対応が必要になることもあった。しかし、ドキュメントをきちんと書いたりそれらのバグに対応したりしているうちに、Equinox チームはいろんな心配ごとを解決できるようになった。今や p2 は、多くの商用製品のプロビジョニングエンジンとして使われている。Eclipse Foundation も、組織でとりまとめたリリースを毎年リリースしているが、このリリースにも p2 集約リポジトリを使っている。これは、すべての寄贈プロジェクトをとりまとめたものだ。

7.4　Eclipse 4.0

アーキテクチャは常に、それで適切かどうかを評価し続けないといけない。新しいテクノロジーを取り込むことができるか? コミュニティの成長を促せるか? 新しい参加者にとって魅力的なものだろうか? 2007 年後半、Eclipse プロジェクトのコミッターたちは、現状ではこれらの問いへの答えが「ノー」であると判断した。そして、Eclipse のビジョンを刷新すべく動き出した。そうは言っても、既存の API に依存する Eclipse プリケーションが大量にあることもわかっていた。2008 年の後半になって、技術評価用のプロジェクトが立ち上がった。その目標は次の三つだ。Eclipse のプログラミングモデルを単純化すること、新たなコミッターを獲得しやすいよう魅力的にすること、そして、ウェブベースのテクノロジーの利点をいかしたプラットフォームをオープンなアーキテクチャとして提供すること。

2010 年 7 月に、Eclipse 4.0 の最初のリリースが公開された。このリリースの狙いは、アーリーアダプターからのフィードバックを得ることだった。3.6 系のリリースに含まれていた SDK バンドルと新しいバンドル群を組み合わせたもので、新しいバンドルは技術評価プロジェクトの成果物だった。3.0 と同様に互換性のためのレイヤーも組み込まれており、既存のバンドルは新しいリリースでも使える。いつものことではあるが、この互換性の恩恵を受けるには、公開 API だけを使ってバンドルを書いておく必要がある。公開 API 以外の内部コードを使っているバンドルについては、互換性は保証されなかった。4.0 のリリースでは Eclipse 4 アプリケーションプラットフォームも含まれており、次のような機能を提供していた。

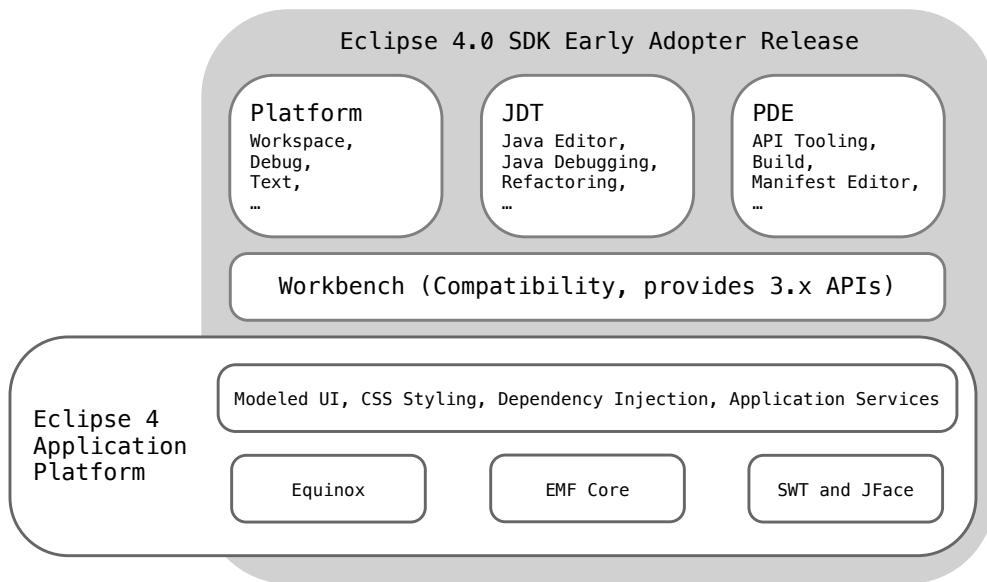



図 7.9: Eclipse 4.0 SDK Early Adopter Release

モデルワークベンチ

4.0 では、Eclipse Modeling Framework (EMFgc) を使ってモデルワークベンチを生成する。これは、ビューのレンダリングとモデルの間で関心の分離を実現するものだ。レンダラーはモデルとやりとりし、そして SWT のコードを生成する。デフォルトでは SWT レンダラーを使うが、他の方法も使える。4.x のアプリケーションを作ると、デフォルトのワークベンチ用に XMI ファイルが作られる。モデルを変更するとワークベンチもすぐに更新され、モデルの変更が反映される。図 7.10 は、サンプルの 4.x アプリケーション用に生成したモデルの例だ。

スタイルシートによる整形

Eclipse が最初にリリースされたのは 2001 年で、当時はまだ、CSS で見た目を切り替えられるようなリッチインターネットアプリケーションの時代ではなかった。Eclipse 4.0 からはスタイルシートを使えるようになり、Eclipse アプリケーションの見た目を簡単に変更できるようになった。デフォルトのスタイルシートは、`org.eclipse.platform` バンドルの `css` フォルダに入っている。

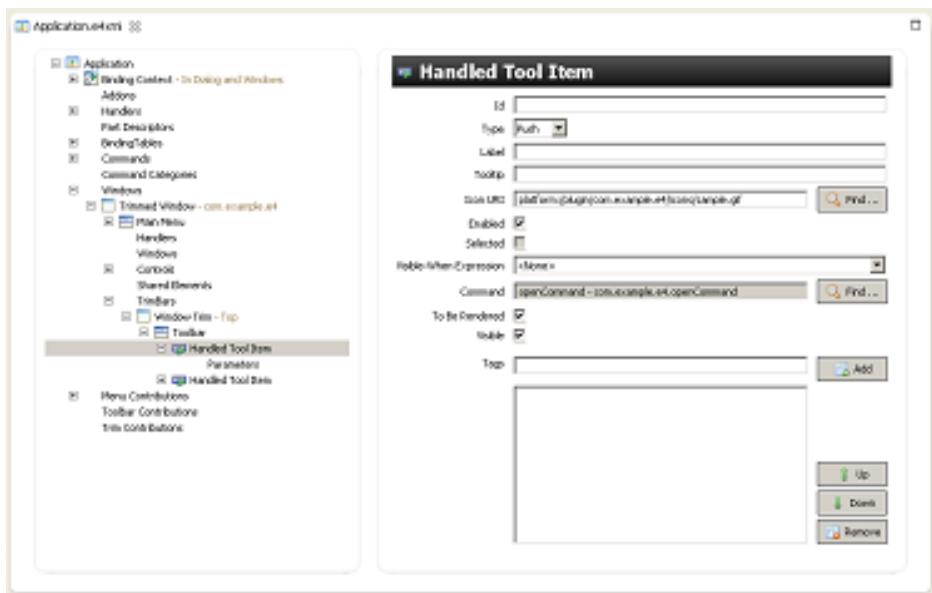



図 7.10: サンプルの 4.x アプリケーション用に生成されたモデル

依存性の注入

Eclipse の拡張リポジトリや OSGi サービスは、サービスプログラミングモデルの一例だ。慣例により、サービスプログラミングモデルには、サービスの提供者（プロデューサー）と利用者（コンシューマー）が存在する。プロデューサーとコンシューマーの間の関係を取り持つのが、ブローカーの役割だ。

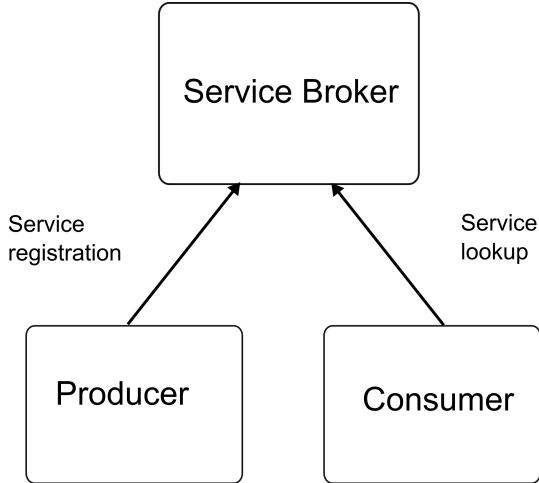



図 7.11: プロデューサーとコンシューマーの関係

伝統的に、Eclipse 3.4.x のアプリケーションでは、コンシューマー側がサービスの場所を知っている必要があった。また、サービスを利用するには、そのサービスのフレームワーク内での継承関係も知っている必要があった。そのため、コンシューマーのコードはあまり使い回しが利かなくなる。というのも、そのコンシューマーが利用するサービスを上書きできないからだ。たとえば、Eclipse 3.x でステータス行のメッセージを更新したいときには、こんなコードを書いていた。

```
getViewSite().getActionBars().getStatusLineManager().setMessage(msg);
```

Eclipse 3.6 はコンポーネントベースで作られてはいるものの、そのコンポーネントの大半は密結合になっていた。より疎結合なコンポーネント群によるアプリケーションを作るために、Eclipse 4.0 では依存性の注入を使ってクライアントにサービスを提供することにした。Eclipse 4.x における依存性の注入は、カスタムフレームワークを使って実現する。このフレームワークは、コンテキストという概念を使ってコンシューマーにサービスの場所を知らせる仕組みを提供する。コンテキストは、アプリケーションとフレームワークの間に位置づけられる。また、コンテキストは階層構造になっている。あるコンテキストが自分には対応できないリクエストを受け取ると、そのリクエストを親コンテキストに委譲する。`IEclipseContext` という名前の Eclipse コンテキストには、利用できるサービスの情報が格納される。そして、OSGi サービスルックアップ機能を提供する。基本的に、コンテキストは Java のマップのようなものだ。名前あるいはクラスとオブジェクトの間のマッピングを行う。コンテキストは、モデルの要素やサービスを処理する。モデルのすべての要素は、コンテキストを持っている。4.0 でサービスを公開するには、OSGi のサービス機能を利用する。

プロデューサーは、サービスやオブジェクトをコンテキストに追加して、そこに格納させる。サービスをコンシューマーオブジェクトに注入するのが、コンテキストの役割だ。コン

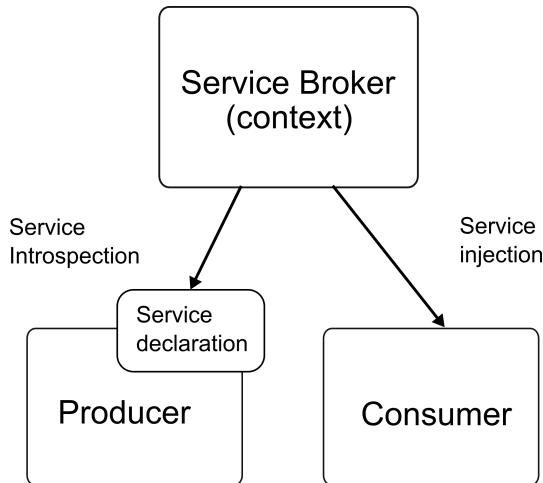



図 7.12: サービスブローカーコンテキスト

シューマーは自分が何を欲しいのかを宣言し、コンテキストがそのリクエストを満たす方法を判断する。この手法のおかげで、動的なサービスを使うのがより簡単になった。Eclipse 3.x では、コンシューマーにリスナーをアタッチして、サービスが使えるようになったときや使えなくなったときに通知を受ける必要があった。Eclipse 4.x では、いったんコンテキストをコンシューマーオブジェクトに注入してしまえば、あらゆる変更が自動的にオブジェクトに配送される。言いかえれば、変更があれば再び依存性の注入を行うということだ。コンシューマーは、自分がコンテキストを使うことを示すために Java 5 のアノテーション機能を利用する。JSR 330 の標準にしたがった`@inject` のようなアノテーションのほかに、Eclipse 独自のカスタムアノテーションも利用する。コンストラクターインジェクション、メソッドインジェクション、そしてフィールドインジェクションに対応している。4.x のランタイムは、これらのアノテーションに沿ってオブジェクトをスキャンする。実際に見つかったアノテーションによって、実行されるアクションが決まる。

　このようにしてコンテキストとアプリケーションの間で関心を分離したおかげで、コンポーネントの再利用がしやすくなった。もはや、コンシューマーがサービスの実装を知っている必要はなくなった。4.x では、ステータス行を更新するコードはこんなふうになる。

```
@Inject
IStatusLineManager statusLine;
...
statusLine.setMessage(msg);
```

アプリケーションサービス

Eclipse 4.0の目標の一つは、利用者向けのAPIをシンプルにして、共通サービスを実装しやすくすることだった。シンプルなサービスの一覧は“the twenty things”と呼ばれ、Eclipseアプリケーションサービスとして知られるようになった。その狙いは、スタンドアローンのAPIを提供して、あらゆるAPIを熟知していなくてもクライアントから利用できるようにすることだ。個別のサービス群として作られており、Java以外の言語、たとえばJavaScriptなどからでも使える。たとえばアプリケーションモデルにアクセスするAPIがあり、設定情報を読み書きしたり、エラーや警告を表示したりできる。

7.5 結論

Eclipseのコンポーネントベースのアーキテクチャは、新しいテクノロジーを取り込みながらも後方互換性を重視して成長してきた。これは容易なことではなかったが、その苦労のおかげでEclipseのコミュニティも成長した。Eclipseベースの製品を作っている人たちに対して、APIが安定しているという安心感を与えることができたからだ。

Eclipseの利用者層はさまざまで、いろんな使い方で利用されている。APIが増えていくにつれて、新しく入ってきた利用者がAPIを把握して使えるようになるまでの敷居が高くなった。今思えば、APIはもっとシンプルにしておくべきだったのだろう。利用者の8割が、APIの2割しか使っていないのだとしたら、APIをもっと単純化する必要がある。Eclipse 4.xでストリームが導入された理由の一つもそれだ。

群衆の叡智のおかげで、想像もしていなかった利用法も出てくるようになった。IDEをいくつかのバンドルに切り分けて、それらを使ったRCPアプリケーションを作るなどといったものだ。逆に、ユーザーからの要求がちょっとしたノイズに感じられるようなこともよくあった。めったに発生しないようなエッジケースに関する要求で、それを実装するにはかなりの時間がかかるといったものだ。

初期のEclipseプロジェクトでは、コミッターは時間をぜいたくに使うことができた。ドキュメントを書いたりサンプルプログラムを書いたり、あるいはコミュニティの質問に答えたりといったことをする時間を存分に確保できていたのだ。時を経て、そういった作業の負荷がEclipseコミュニティ全体にのしかかってきた。ドキュメントや実際の利用例などを用意してコミュニティを支援したいという気持ちはあったものの、リリースのたびに大量の機能追加などが発生する状態では、それも不可能だった。どうせリリース日は遅れるだろうという想定に反して、Eclipseは常にリリーススケジュールを守れていた。そのため、利用者側も、Eclipseはいつも予定通りにリリースされるものだと信用してくれるようになった。

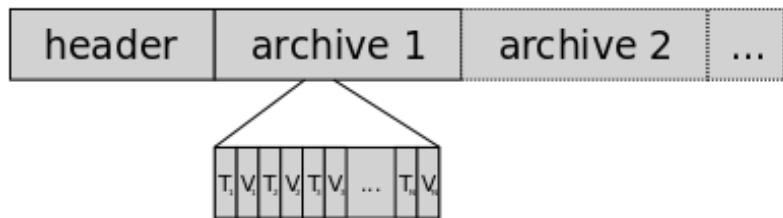
新しいテクノロジーを取り込んだり、Eclipseの見た目や動きを改良したりするためにも、利用者との対話をこれからも大切にしたい。また、いろんな人がコミュニティに参加してくれることを期待する。これを読んでEclipseに興味を持った人は、ぜひhttp://www.eclipse.orgに来てほしい。

第8章

Graphite

Chris Davis

Graphite[1] が行うのはたった二つのとてもシンプルな作業だけ。刻々と変わる値を記録することと、それをグラフ化することだ。同じようなことをするソフトウェアは、これまでにも数多く存在した。Graphite がそれらと一線を画すのは、その機能をネットワークサービスとして提供しているという点だ。そのおかげで、使いやすいだけでなくとてもスケーラブルになる。データを Graphite に渡すときのプロトコルは非常にシンプルで、ほんの数分もあれば手でも書けるようになるだろう (実際に手で書こうとは思わないだろうが、それくらいシンプルだということだ)。グラフをレンダリングしたりデータポイントを取得したりといった操作は URL にアクセスするのと同じくらい簡単である。そのおかげで、Graphite を他のソフトウェアと組み合わせて使うのも簡単だし、Graphite を裏で使った強力なアプリケーションを作ることもできる。Graphite の最もよくある利用例は、ウェブベースのダッシュボードを作ってデータの監視や分析のために使うというものだ。Graphite は大規模な e コマース環境で使うために作られたものであり、それを反映した設計となっている。スケーラビリティがあること、そしてリアルタイムにデータにアクセスできること。これが最大の目標だ。

そのために Graphite が用意したのが、特化型のデータベースライブラリとそれ用のストレージフォーマット、入出力操作の最適化のためのキャッシュ機構、シンプルながらも効率的な方法での Graphite サーバーのクラスタリングといったコンポーネントである。本章では、単に現時点での Graphite の動きを解説するだけでなく、Graphite を最初に書いたときの様子 (どれだけ世間知らずだったか) やどんな問題に遭遇したか、そしてそれをどうやって解決したかなどについても説明する。

8.1 データベースライブラリ: 時系列データの格納

Graphite はすべて Python で書かれており、三つの主要なコンポーネントで構成されている。データベースライブラリの `whisper`、バックエンドデーモンの `carbon`、そしてフロントエン

[1] http://launchpad.net/graphite

ドのウェブアプリケーション。フロントエンドは、グラフのレンダリングをして基本的な UI を提供する。whisper は Graphite 用に書かれたものではあるが、それ単体で独立して使うこともできる。その構造は RRDtool が使うラウンドロビンデータベースと似ており、時系列の数値データだけを格納する。通常はデータベースをサーバープロセスと考え、クライアントアプリケーションはそれに対してソケット越しにやりとりする。しかし whisper は、RRDtool と同様にアプリケーションが利用するデータベースライブラリであり、特別なフォーマットのファイルに保存されたデータを扱うものだ。whisper の基本操作には、新たな whisper ファイルを作る create や新しいデータポイントをファイルに書き込む update、そしてデータポイントを取得する fetch がある。

図 8.1: whisper ファイルの基本構造

図 8.1 に示すとおり、whisper ファイルにはヘッダセクションがあってそこにさまざまなメタデータが格納される。そしてヘッダセクションの後にはいくつかのアーカイブセクションが続く。各アーカイブは一連のデータポイント群で構成されており、データポイントは (timestamp, value) のペアとなる。update あるいは fetch を実行すると、whisper はまずデータの読み書きをするファイル内のオフセットを調べる。このときに利用するのは、タイムスタンプとアーカイブ構成だ。

8.2 バックエンド: シンプルなストレージサービス

Graphite のバックエンドは carbon-cache という名前のデーモンプロセスで、通常は carbon と呼ばれている。このバックエンドは Twisted を使って作られている。Twisted とは、Python 用に書かれた高度にスケーラブルなイベント駆動の I/O フレームワークである。Twisted のおかげで、carbon は大量のクライアントとのやりとりをこなせ、大量のトラフィックをさばいてもオーバーヘッドは少なくて済む。図 8.2 は、carbon と whisper そしてウェブアプリケーションの間でのデータの流れを表したものである。クライアントアプリケーションが収集したデータを Graphite のバックエンドである carbon に送信し、そのデータは whisper を使って格納する。このデータを Graphite ウェブアプリケーションが活用して、グラフを生成する。

carbon の主機能は、クライアントから渡されたメトリクスをデータポイントとして格納することである。Graphite における「メトリクス」とは、計測可能な量であって時間とともに

図 8.2: データの流れ

変化するもの (サーバーの CPU 使用率や商品の売上数量など) を意味する。データポイントとは単なる (timestamp, value) のペアであり、ある時点での特定のメトリクスの計測値に対応する。メトリクスには一意な名前がついており、メトリクスの名前とそのデータポイントがクライアントから送られてくる。クライアントアプリケーションとしてよくある形式は、何らかのシステムやアプリケーションのメトリクスを監視するエージェントとして働くものである。収集した値を carbon に送り、データの格納と可視化をお手軽に行う。Graphite のメトリクスにはシンプルな階層型の名前がついている。ファイルシステムのパスと似たものだが、階層の区切りにはスラッシュやバックスラッシュではなくドットを使う。carbon は正当な名前ならすべて受け入れ、whisper ファイルをメトリクスごとに作ってそこにデータポイントを格納する。whisper ファイル群は carbon のデータディレクトリに格納される。そのディレクトリ構成は、メトリクスの名前をドットで区切った階層を反映したものとなる。つまり、たとえば servers.www01.cpuUsage なら.../servers/www01/cpuUsage.wsp に格納されることになる。

クライアントアプリケーションからデータポイントを Graphite に送信するには、まず carbon との TCP コネクションを確立する必要がある。通常は、ポート 2003 を利用する [2]。話しかけるのはクライアント側だけで、carbon がネットワーク越しに何かを送信することはない。クライアントはデータポイントをシンプルなプレーンテキスト形式で送信し、コネクションは開いたままにしておいて必要に応じて再利用する。そのフォーマットはデータポイントご

[2] もうひとつ別のポートもあって、シリアライズしたオブジェクトを送れるようにもなっている。そのほうが、プレーンテキスト形式で送るよりも効率的になる。ただ、これが必要となるのはトラフィックが非常に激しい場合だけである。

とに 1 行のテキストにまとめたものであり、各行にはドット区切りのメトリクス名とその値、そして Unix タイムスタンプがスペース区切りで書かれている。たとえば、クライアントから送るデータの形式はこのようになる。

```
servers.www01.cpuUsage 42 1286269200
products.snake-oil.salesPerMinute 123 1286269200
[one minute passes]
servers.www01.cpuUsageUser 44 1286269260
products.snake-oil.salesPerMinute 119 1286269260
```

上位レベルでとらえた carbon の仕事は、この形式のデータを待ち受けて、受け取ったデータを即時に whisper でディスクに書き込むというのがすべてだ。本章では後ほど、スケーラビリティを確保しつつ一般のハードディスクドライブで最高のパフォーマンスを引き出すための小技についても紹介する。

8.3 フロントエンド: グラフオンデマンド

Graphite のウェブアプリケーションを使えば、シンプルな URL ベースの API を使ってカスタムグラフを作らせることができる。グラフの各種パラメータの指定には HTTP GET リクエストのクエリ文字列を使い、レスポンスとして PNG 画像が戻ってくる。たとえば、次の例を考えよう。

```
http://graphite.example.com/render?target=servers.www01.cpuUsage&
width=500&height=300&from=-24h
```

この URL へのリクエスト結果は 500×300 のグラフとなる。メトリクス servers.www01.cpuUsage の、過去 24 時間のデータを描画したものだ。実際のところ、必須パラメータは target だけであり、それ以外はすべて任意指定だ。省略した場合はデフォルト値を利用する。

Graphite はさまざまな表示オプションに対応しており、シンプルな関数型の構文によるデータ操作関数も用意されている。たとえば、先ほどの例において 10 ポイント間の移動平均をグラフにしたければ次のようにすればよい。

```
target=movingAverage(servers.www01.cpuUsage,10)
```

関数はネストできるので、複雑な表現や計算も可能である。

もうひとつ別の例を示す。これは、その日の現時点までの売上合計を、商品ごとの毎分売上数量を使って算出している。

```
target=integral(sumSeries(products.*.salesPerMinute))\&from=midnight
```

`sumSeries` 関数は、パターン `products.*.salesPerMinute` にマッチする各メトリクスの合計について時系列データを計算する。そして `integral` が、毎分のカウントではなくそこまでの合計を計算する。ここまでくれば、グラフを表示したり操作したりするウェブ UI を作るのもそれほど難しくはなさそうだと思えるだろう。Graphite には自前の Composer UI が組み込まれている。図 8.3 のようなもので、ユーザーがメニューから何かの機能をクリックすると JavaScript を使ってグラフの URL パラメータを変更する。

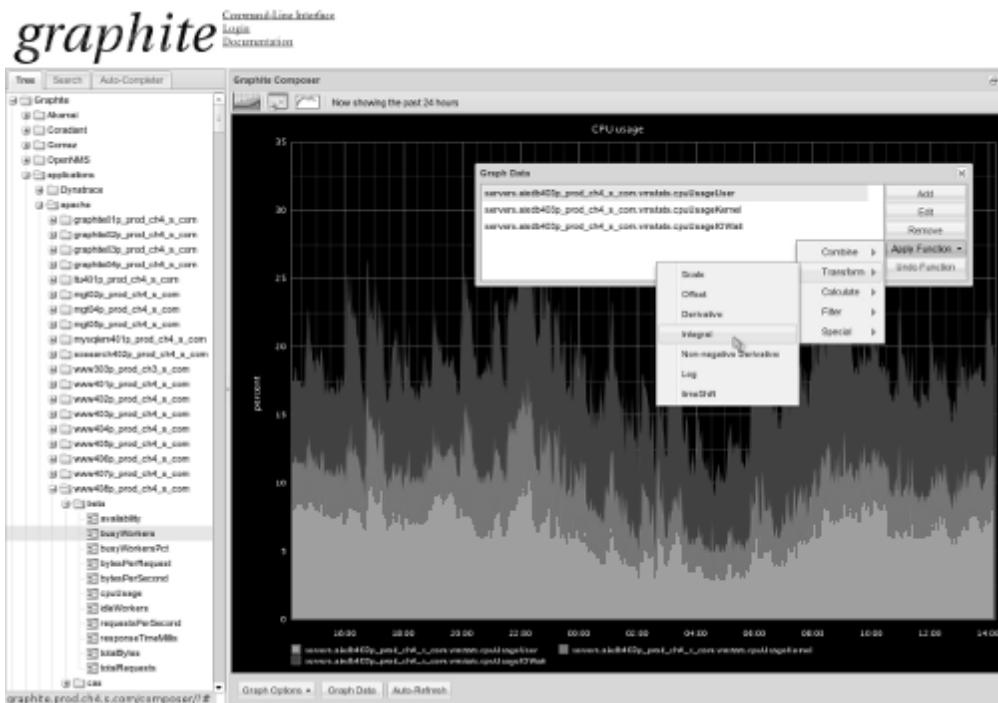



図 8.3: Graphite の Composer Interface

8.4　ダッシュボード

その黎明期から、Graphite はウェブベースのダッシュボードを作るためのツールとして使い続けられてきた。URL 形式の API があるので、これはごく自然な使い方である。ダッシュボードの作り方はとても単純で、単にこんな感じのタグを使った HTML ページを作るとの同じ程度の話だ。

```
<img src="http://graphite.example.com/render?parameters-for-my-awesome-graph">
```

しかし、URL を手で組み立てるのが気に入らない人もいるだろう。そこで、Graphite の Composer UI ではマウスのクリックだけでグラフを作れる仕組みを用意した。グラフができあがれば、あとはその URL をコピーして使うだけで済む。ウェブページを作成するためのさまざまなツール (Wiki など) と組み合わせれば、技術に詳しくないユーザーでも自分用のダッシュボードが簡単に作れるようになる。

8.5　明らかなボトルネック

あるユーザーがダッシュボードを作り始めたとき、Graphite があっという間にパフォーマンスを落としてしまった。私はウェブサーバーのログを調べ、いったいどんなリクエストが原因なのかを確かめた。その結果わかったのは、グラフ描画のリクエストが大量に発生しているのが原因だということだった。絶え間なくグラフのレンダリングをしていたので、CPU がウェブアプリケーションのボトルネックとなっていたのだ。まったく同じリクエストが大量に発生していることに気付き、ダッシュボードに原因があることがわかった。

ダッシュボードに 10 個のグラフが表示されており、それが 1 分おきに再読み込みされる様子を想像してみよう。ユーザーがダッシュボードをブラウザで開くたびに、Graphite は毎分 10 リクエストを余分に処理しなければいけなくなる。あっという間に負荷は上がってしまう。

シンプルな解決策は、それぞれのグラフのレンダリングを一度だけにして、各ユーザーにはそのコピーを返すようにするという方法だ。Graphite が利用しているウェブフレームワークの Django にはすばらしいキャッシュ機構が組み込まれており、memcached などのさまざまなバックエンドを使うようにできる。memcached[3] は、本質的にはハッシュテーブルをネットワークサービスとして提供する仕組みである。クライアントアプリケーションからキー・バリューのペアを取得したり設定したりなど、ふつうのハッシュテーブルと同じことができる。memcached を使う最大の利点は、重たいリクエスト (グラフのレンダリングなど) の結果を手早く保存してそれ以降のリクエストで流用できることだ。最新状態を反映していないグラフをいつまでも返し続けることを避けるために、キャッシュしたグラフの有効期限を memcached の設定で短めにしておくことができる。友好期限をほんの数秒にしておいたとしても、Graphite への負荷は大きく下がる。リクエストの重複があまりにも頻発しているからだ。

レンダリングのリクエストが大量に発生する場面がもうひとつ存在する。Composer UI 上で、表示オプションをいじったり関数を適用したりするときだ。ユーザーが何かを変更するたびに Graphite はグラフを再描画しなければいけない。個々のリクエストは同じデータに関するものなので、もとになるデータを memcache に入れておくと効率がよい。こうしておけば、ユーザーに反応を返すのが遅れずに済む。データの取得処理を飛ばせるからだ。

---

[3] http://memcached.org

8.6　入出力の最適化

60,000 件のメトリクスを Graphite サーバーに送信し、そのそれぞれが 1 分ごとにデータポイントを作るという場面を考えてみる。各メトリクスにに対応する whisper ファイルがファイルシステム上に用意されていたことを思い出そう。つまり、carbon は 60,000 の異なるファイルに毎分 1 回の書き込みをする必要があるということだ。carbon が 1 ファイルへの書き込みを 1 ミリ秒で行えるのなら、十分に対応できる。これくらいならそんなに無茶な話ではない。でも、毎分更新するメトリクスが 600,000 件あったとしたらどうだろう? あるいは更新頻度が 1 秒ごとだったら?高速なストレージを導入する余裕がなかったとしたら? まあなんでもいい。入力のデータポイントの分量が、ストレージ側の書き込み速度の限界に達してしまったとしよう。そんな場合はどうすればいいだろうか?

最近のハードディスクドライブは、シークタイムが遅いものが多い[4]。そのため、別々の場所への書き込みのほうがシーケンシャルな書き込みよりも遅延が大きくなる。つまり、連続書き込みを増やせば増やすほど、スループットを稼げるというわけだ。しかし、数千ものファイルに頻繁に書き込む必要があって個々の書き込みの分量が非常に小さい場合 (whisper のデータポイントは 1 件あたりたったの 12 バイトだ) は、結局のところは大半の時間をシークに費やすことになる。

書き込み操作の比率が比較的早めに限界に達するという前提で考えると、その限界を超えるスループットをたたきだすための唯一の方法は、複数のデータポイントの書き込みを一度の書き込み操作で済ませることだ。これは実現可能である。なぜなら whisper は、連続したデータポイントをディスク上でも連続した領域に配置するからだ。というわけで、私は whisper に update_many 関数を追加した。この関数は、ひとつのメトリクスに関するデータポイントのリストを受け取り、連続したデータポイントを単一の書き込み操作に圧縮する。一回の書き込みのデータ量は大きくなってしまうが、10 件のデータポイント (120 バイト) の書き込みと 1 件のデータポイント (12 バイト) の書き込みに要する時間の差は無視できる程度のものである。よっぽど大量のデータポイントをひとまとめにしない限り、サイズの差がレイテンシに影響を及ぼすことはないだろう。

次に私は、carbon にバッファリングの機能を実装した。入力のデータポイントをメトリクス名に基づくキューにマップし、対応するキューにデータポイントを追加していったのだ。別スレッドから繰り返し全キューを走査し、すべてのデータポイントを取得する。そして、適切な whisper ファイルに update_many で書き込んでいった。先ほどの例に戻ろう。600,000 件のメトリクスを 1 分ごとに更新するときに、ストレージ側では 1 回の書き込みに 1 ミリ秒かってしまうとする。そんな場合なら、平均して 10 件程度のデータポイントを各キューに持たせることになる。このとき使用量が増える唯一のリソースはメモリであり、メモリは比較的潤沢に使える。というのも各データポイントはたかだか数バイトにしかならないからだ。

この方式では、必要に応じてデータポイントを動的にバッファリングし、ストレージ側の I/O が持ちこたえられる程度に入力データポイントのスピードを抑えることになる。この方

[4]SSD なら、一般的にハードディスクドライブよりもずっとシークタイムが高速になる。

式のよいところは、一時的に I/O がスローダウンしたときなどに弾力的な扱いができるところだ。Graphite 以外の他の場所で I/O を伴う作業が発生するなら、書き込み操作を減らせばよい。その場合は単に carbon のキューが大きくなるだけのことだ。キューが大きくなればなるほど、書き込みの量も大きくなる。データポイントの全体的なスループットは書き込み操作の回数と各書き込みの平均サイズを掛け合わせたものになるので、キュー用に確保できるメモリが十分にある限りは carbon を持ちこたえさせることができる。carbon のキューの仕組みは図 8.4 のようになる。

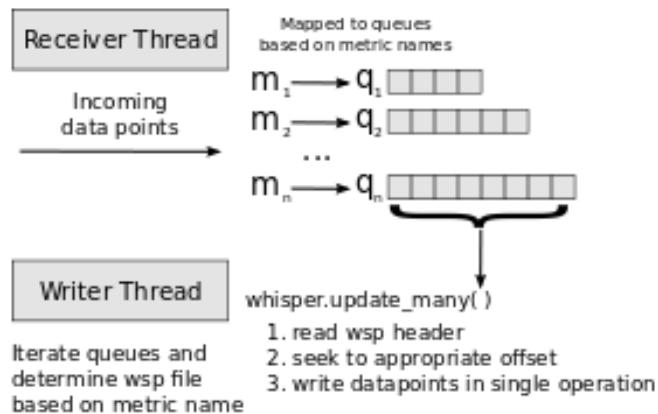



図 8.4: Carbon のキューのしくみ

8.7 リアルタイム性の維持

データポイントをバッファリングするというのは carbon の I/O の最適化という面ではよい方法だ。しかしすぐに、それ以外の副作用も見つかってしまった。ここでもう一度、先ほどの例に戻ってみよう。600,000 件のメトリクスを 1 分ごとに更新したいけれど、ストレージの性能的には 1 分間に 60,000 回の書き込み操作が限界である、という状況だ。これはつまり、どの時点を見ても carbon のキューにはほぼ 10 分ぶんのデータが溜まっているということになる。ユーザーの視点で考えると、Graphite のウェブアプリケーションからリクエストを送っても最新の 10 分のデータは表示されないということだ。これはひどい!

幸いなことに、この問題は簡単に解決できる。私は carbon にソケットリスナーを追加して、バッファリングされたデータポイントにアクセスするための問い合わせインターフェイスを作った。そして Graphite のウェブアプリケーションを修正し、データを取得するときにはこのインターフェイスを使うようにしたのだ。このインターフェイスで carbon から取得したデータポイントとディスクから取得したデータポイントを組み合わせれば、ほらできあがり。これでリアルタイムのグラフを表示できるようになった。確かに、先ほどの例は 1 分

ごとの更新だったし厳密には「リアルタイム」だとは言えないかもしれない。しかし実際のところ、carbon が受け取ったデータポイントが即時にグラフに反映されるという意味ではリアルタイムだと言えるだろう。

8.8　カーネル、キャッシュ、壊滅的な障害

そろそろお気づきだろうが、Graphite のパフォーマンスに大きな影響を及ぼすのは I/O のレイテンシーである。ここまでは、I/O のレイテンシーが書き込みあたり 1 ミリ秒という比較的低めの環境を想定してきた。しかしこれはおおざっぱな見込みにすぎず、もう少し深く分析する必要がある。大半のハードディスクドライブは、そんなに速度が出ない。10 台以上のディスクで RAID アレイを組んだとしても、ランダムアクセスのレイテンシーはきっと 1 ミリ秒を超えることだろう。しかし、たとえば旧型のラップトップで 1 キロバイトのデータをディスクに書き込んだ場合、write システムコールは 1 ミリ秒よりずっと速く結果を返すはずだ。いったいなぜだろう?

あるソフトウェアが一見してつじつまの合わない予想外のパフォーマンスを見せた場合、たいていはバッファリングあるいはキャッシュがからんでいる。この場合は、その両方だ。write システムコールはデータを実際にディスクに書き込むわけではなく単にバッファに入れるだけであり、後でディスクに書き込むのはカーネルの役割となる。だからこそ、write が予想外に速く処理を終えるのだ。バッファをディスクに書き込んでからも、それ以降の read に備えてキャッシュしたままにしておくことが多い。こういったバッファリングやキャッシュを行うには、当然ながらメモリが必要になる。

賢いカーネル開発者たちは、ユーザーメモリ空間で余っているメモリを使うほうがきっちりメモリを確保しておくよりもよさげだと判断した。そのおかげでパフォーマンスを大きく確保できたわけだ。システムのメモリをいくら増やしても I/O を繰り返すうちにいつの間にか “空き” メモリがゼロになっている、という現象もこれで説明がつく。ユーザー空間のアプリケーションでそんなにメモリを使っているわけでもないのになあ…というときは、おそらくカーネルがメモリを使っているのだろう。この手法には弱点もある。ユーザー空間のアプリケーション側で必要なメモリが増えるとカーネルが使える “空き” メモリがなくなってしまうということだ。そんな場合はカーネル側でメモリを使うのをあきらめざるを得ず、それまで保持していたバッファの中身を失ってしまう。

そんなこと、Graphite に何の関係があるんだって? 先ほど強調したように、carbon は I/O のレイテンシが一貫して低いことを前提としている。また、write システムコールが高速に動作するのは単にデータをバッファにコピーしているだけだからだということも知っている。カーネルが使えるメモリが少なくなって、バッファへの書き込みが続行不能になったら、いったいどうなるだろう? 同期書き込みしかできなくなるわけで、これは恐ろしく遅い! これは carbon の書き込み操作にも深刻な影響を与え、その結果 carbon のキューが肥大化する。その結果としてメモリの使用量が増え、カーネルが使えるメモリはさらに減って……。最終的

にどうなるか。carbon がメモリを食いつぶしてしまうか、いらついたシステム管理者に強制終了させられてしまうだろう。

こんな悲惨な結果を避けるために、私は carbon にちょっとした機能を追加した。キューに格納できるデータポイント数や whisper の各種操作の頻度を、設定で制限できるようにしたのだ。これらの機能を使えば、carbon が制御不能なスパイラルに陥ることを防げる。そのかわりにいくつかのデータポイントを取りこぼしたりデータポイントをさばききれなくなったりといった影響も出るが、そうしないときの悪影響に比べればまだ我慢できる。しかし、これらの適切な設定値はシステムによって異なり、何度もテストしないと適切な値にチューニングできないだろう。便利な機能ではあるが、問題を根本的に解決するというものではない。根本的に解決するには、ハードウェアを増強するしかない。

8.9　クラスタリング

複数の Graphite サーバーを立てつつ、ユーザー側からは単一のシステムに見えるようにする。そんなことも、対して難しくはない。少なくとも、馬鹿正直な実装である限りは。Graphite ウェブアプリケーションのユーザーインターフェイスは、主に二種類の操作で成り立っている。メトリクスの選択 (find) と、(通常はグラフ形式での) データポイントの取得 (fetch) だ。これらの操作はライブラリとして切り出されており、その実装が他のコードベースとは切り離されて抽象化されている。また、HTTP リクエストハンドラも公開しており、リモートからの呼び出しにも対応できる。

find 操作は、ローカルファイルシステム上の whisper データを検索して、ユーザーが指定したパターンに一致するものを見つける。ちょうど、ファイルシステム上で*.txt などとして特定の拡張子のファイルを探すのと同じようなことだ。find が返す結果はツリー構造で Node オブジェクトのコレクションとなり、Node のサブクラスである Branch あるいは Leaf のいずれかで表される。ディレクトリがブランチノードに、そして whisper ファイルがリーフノードにそれぞれ対応する。この抽象化レイヤーのおかげで、ベースとなるストレージをさまざまな形式に対応させるのも簡単になった。たとえば RRD ファイル[5] や gzip した whisper ファイルにも対応している。

Leaf インターフェイスには fetch メソッドが定義されており、その実装はリーフノードの形式によって異なる。whisper ファイルの場合は、単に whisper ライブラリ自体の fetch 関数を薄くラップしているだけである。クラスタリングに対応し始めたときに find 関数は拡張され、HTTP 経由でリモートの find 関数を呼べるようになった。呼び出す先の Graphite サーバーは、ウェブアプリケーションの設定項目として指定する。ノードのデータには、これらの HTTP コールの結果を RemoteNode オブジェクトでラップしたものが含まれるようになる。このオブジェクトは、通常の Node や Branch そして Leaf といったインターフェイスに対応している。このおかげで、クラスタリング環境をウェブアプリケーションで透過的に扱える

[5]実は、RRD ファイルはブランチノードである。というのも、ひとつのファイルに複数のデータソースを含めることができるからだ。RRD のデータソースがリーフノードとなる。

ようになった。リモートリーフノードの fetch メソッドは別の HTTP コールとして実装されており、データポイントをそのノードの Graphite サーバーから取得するようになっている。

ウェブアプリケーション間でのこれらの呼び出しは、すべてクライアントからの呼び出しと同じように行っている。ただ、追加のパラメータがひとつあって、その操作をローカルだけで行いクラスタには伝搬させないという指定もできる。グラフを描画するよう指示を受けたウェブアプリケーションは、まず find を使って対象のメトリクスを探し、そして fetch を呼んでそのデータポイントを取得する。データがローカルサーバーにあろうがリモートサーバーにあろうが、あるいはその両方に分かれていようがこの動きは変わらない。サーバーがダウンしている場合、リモート呼び出しがタイムアウトするとすぐにそのサーバーをサービス不能状態と認識し、一定の期間はそのサーバーへの呼び出しが発生しないようになる。ユーザー側から見れば、ダウン中のサーバーにあったデータはグラフから抜けた状態になる。ただし、クラスタ内の他のサーバーにデータのコピーがある場合はその限りではない。

クラスタリングの効率についてのちょっとした分析

グラフを表示するリクエストの中で最もコストのかかる部分は、グラフのレンダリングである。レンダリングは単一のサーバーで行われるので、サーバーを増やせばグラフのレンダリング能力が向上する。しかし、リクエストの結果として発生する find の呼び出しがクラスタ内のあらゆるサーバーに向かうので、フロントエンドにかかる負荷の多くを各サーバーで共有することになってしまう。しかし、既に我々はバックエンドの負荷をうまく分散させる方法を手に入れている。個々の carbon インスタンスを独立して動かすことである。これは、はじめの一歩としてはよい方法だ。というのも、ほとんどの場合はフロントエンドよりバックエンドのほうが先にボトルネックになるからである。しかし、当然ながら、この方式ではフロントエンドをスケールアウトさせることができない。

フロントエンドをもっとうまくスケールさせるには、ウェブアプリケーションから発行するリモートへの find の呼び出しを減らす必要がある。一番簡単な解決法は、毎度おなじみのキャッシュだ。既に memcached を導入してデータポイントやレンダリング済みグラフをキャッシュしているが、さらに find の結果もここにキャッシュしておける。各メトリクスの場所はそんなに頻繁には変わらないので、これは少し長めにキャッシュしておける。しかし、find の結果のキャッシュ期間を長めに設定するトレードオフとして、新しく追加されたメトリクスはしばらく待たないとユーザーに見えなくなってしまう。

クラスタ内での分散メトリクス

Graphite ウェブアプリケーションはクラスタ内のいたるところで均質であり、どのサーバー上でもまったく同じ処理をこなせる。しかし、carbon の役割はサーバーによって異なり、どのデータをどのサーバーに送信するかによって大きく変わってくる。たくさんの異なるクライアントから carbon にデータが送られてくることが多いので、個々のクライアントの設定を

Graphite クラスタ上の配置を関連付けるのはとても面倒な作業だ。アプリケーションのメトリクスは単一の carbon サーバーにまとめ、業務メトリクスは複数の carbon サーバーに送って冗長性を確保するなどということになる。

そういった場合の管理をシンプルにするために、Graphite には carbon-relay というツールも用意されている。このツールの役割は極めてシンプルだ。クライアントからのメトリクスのデータを標準の carbon デーモン (実際の名前は carbon-cache) とまったく同じ方法で受け取るが、そのデータをただ格納するのではなく、メトリクス名に対していくつかのルールを適用して、データをどの carbon-cache サーバーに転送するかを判断する。このときのルールは、正規表現と対象サーバーリストで構成されている。データポイントを受信するたびにルールを順に評価し、正規表現にマッチした最初のルールを利用する。クライアント側で必要なことは、データを carbon-relay に送信するだけだ。その先は、自動的に正しいサーバーに送られるようになる。

ある意味では carbon-relay がレプリケーション機能を提供しているとも言えるが、より正確には入力の複製機能とでも言うべきだろう。というのも、同期機能は持っていないからだ。あるサーバーが一時的にダウンしたとすると、その期間のデータポイントは失われるが、それ以外の機能は正常に動作する。また、システム管理者が再同期処理をコントロールするための管理スクリプトも用意されている。

8.10 設計に関する想い

これまで Graphite に関わってきて、私の認識が正しいことを再確認した。スケーラビリティを確保するには、低レベルのパフォーマンスをいくら改善してもほとんど効果はなく、結局は全体的な設計が重要であるということだ。これまでにいろんなボトルネックが見つかったが、そのたびに考えたのは、パフォーマンスを確保するために速度を上げるということではなく設計を改善させるということだった。「なんでまた、Graphite を Python で書いてしまったんだろう。Java とか C++にすればよかったんじゃない?」と自問することも多かったが、結論はいつも同じ。「Java や C++じゃないとどうしようもないほどのパフォーマンスを要求される場面なんかいままでになかったんだし、それなら Python の使いやすさをとったほうがいいんじゃない?」かのドナルド・クヌースが [Knu74] で言うとおり、早まった最適化は諸悪の根源である。自分のコードが今後も大きく成長し続けるという前提で考える限り、あらゆる最適化 [6] はある意味「早まった」ものだと言える。

Graphite の最大の強みであり、かつ最大の弱みでもあるのが、一般に言われているような意味での “設計” を実はほとんどしていないという事実である。Graphite は徐々に成長してきた。何か問題が発生するたびに、そのハードルをひとつひとつ克服していったという感じだ。数々のハードルの中には事前に予測できるものもたくさんあったし、あらかじめそれらに手を打っておくというのもごく自然に思える。しかし、まだ実際に遭遇していない問題に

[6] クヌースが言及していたのは低レベルのコードの最適化についてであり、設計の改善などの巨視的な最適化を否定しているわけではない。

対応するというのは避けたかった。たとえいずれは対処せざるを得ない問題だったとしても。その理由は、実際に問題に遭遇してから対応するほうが事前に理屈で考えた対応方法よりもうまくいくだろうと考えたからである。問題を解決するときに使ったのは、手元にある検証データと我々の知識や直感だった。自分の知識を疑い続け、実際の検証データをより活用するようにした。

最初に whisper を書いたころは、スピードを考慮すると最終的には C で書きなおすことになるだろうと信じ切っていた。いま書いている Python 版はあくまでもプロトタイプに過ぎないと考えていたのだ。時間に迫られていなければ、Python でのプロトタイプを作ろうなどとは思わなかっただろう。しかし、I/O のボトルネックのほうが CPU よりもずっと早期に顕在化するということがわかった今では、Python を使い続けても事実上問題ないであろうと言える。

しかし、先ほども言ったとおり、この漸進的な手法は Graphite の弱点でもある。インターフェイスは、徐々に成長させていくということができなかった。よいインターフェイスは一貫性があるものであり、予想に反する動きを抑えてできるだけ期待通りに動くような規約を使うものである。この基準に照らし合わせると、Graphite の URL API のインターフェイスは現状では及第点に達していないと思う。オプションや関数がたびたび追加されたため、部分的には一貫しているところもあるけれども全体的に見ればまったく一貫性に欠けるものとなってしまった。この問題を解決するにはインターフェイスをバージョン管理していくしかないが、そうしたとしても問題は残る。新たなインターフェイスを設計したとしても古いものもまだ捨てきれず、どんどん荷物が増え続けてしまう。まるで盲腸のようなものだ。最初のうちはそんなに気にならないかもしれない。しかし、そのうちいつか、コードの虫垂炎 (古いインターフェイスに起因するバグ) を手術するときがやってくる。過去にさかのぼって Graphite を一か所だけ変更できるなら、外部向け API の設計をもっと念入りにやり直したい。その場しのぎで対応するのではなく、事前にきちんと考えておくべきだった。

Graphite でちょっと気になるところがもうひとつある。メトリクスの命名規則が階層型であるせいで、柔軟性に欠けるという点だ。極めてシンプルだしたいていの場合は便利に使えるのだが、込み入ったクエリの表現が非常に難しくなったり、時には不可能になったりしてしまう。Graphite を作ろうと考え始めたころからずっと思っていたのだが、私が欲しいのは、人間が手で編集できる形式の URL を使った API によるグラフの作成機能だった [7]。今の Graphite はその当時の私の思った通りの機能を提供できている。だが、そのせいで API をシンプルにせざるを得ず、複雑な表現をしようとすると手に負えなくなっているのではないかという恐れもある。階層型にしたおかげで、メトリクスの “主キー” を極めてシンプルに考えられるようになった。パス自体が、本質的にそのツリー内でのノードの主キーとなるからである。問題は、説明用のデータ (カラムのデータなど) すべてを直接パスに埋め込む必要があるという点だ。考えられる解決策としては、階層型のモデルとは別にメタデータのデータベースを用意して、より込み入った選択をするための特別な構文を作ることだろう。

[7] そのため、グラフ自体はオープンソースにせざるを得なかった。グラフの URL を見れば何をやっているかは丸わかりだし、手で書き換えるのも簡単だ。

8.11　そしてオープンソースへ…

Graphite のこれまでを振り返ると、プロジェクトとしてここまで成長したことにも驚かされるし、私自身がプログラマーとしてどれだけ成長したかにも驚かされる。もともとは個人的に始めたちょっとしたプロジェクトで、最初のころはたかだか数百行のコードでしかなかったんだ。レンダリングエンジンだって、自分にそんなものが書けるのかを確かめるためのちょっとした実験程度のものだった。`whisper` を書いているときは、公開予定日の直前になって発生した重大な問題に対応するため、週末に泣きながら作業を進めたこともあった。`carbon` は何度となくゼロから書き直すことになった。もはや数え切れないくらいの書き直しをしている。Graphite をオープンソースライセンスで公開する許可を得たのは 2008 年だったが、当時はそんなに反応があるわけがないと思っていた。それから数カ月の間に、CNET の記事で取り上げられたり Slashdot で話題になったりして、いまだかつてないほど活発に動き出した。今や、大企業や中小企業も含めて何十社もが Graphite を使っている。そのコミュニティはとても活発であり、今も成長し続けている。もはや終わってしまったプロダクトとは違って今でも実験的な機能をどんどん取り入れたりしているので、このプロジェクトでの作業は楽しみと可能性に満ちている。

第9章

The Hadoop Distributed File System

Robert Chansler, Hairong Kuang, Sanjay Radia,
Konstantin Shvachko, and Suresh Srinivas

Hadoop Distributed File System (HDFS) は、大規模なデータセットを高い信頼性で格納するために作られたシステムである。また、そのデータセットを広帯域でユーザーアプリケーションに流せるようにもなっている。大規模なクラスター環境では、数千台ものサーバーが自身にアタッチされたストレージを管理したりユーザーアプリケーションのタスクを実行したりする。ストレージや計算処理を多くのサーバーに分散させることで、どんな規模であっても必要に応じてリソースを成長させ続けられ、かつ無駄なく使うことができる。本章では HDFS のアーキテクチャについて解説し、HDFS で 40 ペタバイトものエンタープライズデータを管理する Yahoo!の例を紹介する。

9.1　はじめに

Hadoop[1] が提供するのは分散ファイルシステムであり、さらに、大規模なデータセットを MapReduce [DG04] のパラダイムで分析したり変換したりするフレームワークも用意している。HDFS のインターフェイスは Unix のファイルシステムのパターンに従っているが、アプリケーション側でのパフォーマンスを向上させるためなら標準の規約から外れることもいとわない。

Hadoop の重要な特徴は、データや計算処理を多数の (数千台もの) ホストに分散できることであり、アプリケーションから実行した計算処理をデータ側で並列実行できることである。Hadoop クラスターで計算能力やストレージそして I/O の帯域を拡張するには、そこらにあるごく普通のサーバーを追加するだけでかまわない。Yahoo!の Hadoop クラスターをすべて合わせると 40,000 台のサーバーで構成されており、40 ペタバイトのデータを管理している。最大のクラスターは 4000 サーバーである。それ以外にも、世界中で 100 社以上が Hadoop を採用しているという報告がある。

[1] http://hadoop.apache.org

HDFS は、ファイルシステムのメタデータとアプリケーションのデータを分けて格納する。PVFS [CIRT00] や Lustre[2] そして GFS [GGL03, MQ09] といった他の分散ファイルシステムと同様、HDFS もメタデータは専用のサーバーに格納しており、これを NameNode と呼んでいる。一方、アプリケーションデータを格納する他のサーバー群は DataNodes と呼んでいる。すべてのサーバーが完全に接続されており、お互いのやりとりには TCP ベースのプロトコルを利用する。Lustre や PVFS とは異なり、HDFS の DataNodes は RAID などのデータ保護機構を使わずにデータを永続化する。そのかわり、GFS と同様に、ファイルの内容を複数の DataNode にレプリケートして信頼性を確保する。この方式には、データの永続性を確保するだけでなくそれ以外の利点もある。データ転送の帯域が拡大し、計算に必要なデータに近い場所で計算をできる可能性が高まるのだ。

9.2　アーキテクチャ

NameNode

HDFS の名前空間は、ファイルとディレクトリの階層構造になっている。ファイルとディレクトリは、NameNode 上の inode で表される。inode には、各種の属性が記録されている。パーミッション・変更時刻・アクセス時刻・名前空間・ディスクスペース上の割り当て領域などである。ファイルの中身は大きめのブロック (一般的には 128 メガバイトだが、ファイルごとに指定可能) に分割されており、各ブロックを複数 (一般的には 3 つだが、ファイルごとに指定可能) の DataNode へと個別にレプリケートできる。NameNode では名前空間ツリーを管理し、ブロックと DataNode の対応も管理する。現時点の設計では、クラスターごとにひとつの NameNode を保持している。ひとつのクラスターで何千もの DataNode と何万もの HDFS クライアントを保持でき、それぞれの DataNode が複数のアプリケーションタスクを並列実行できる。

イメージおよびジャーナル

inode、そして名前システムのメタデータを定義するブロックのリストのことを**イメージ**と呼ぶ。NameNode は、名前空間全体のイメージを RAM に保持する。イメージの永続レコードをその NameNode のローカルにあるネイティブファイルシステム上に格納したものをチェックポイントと呼ぶ。NameNode は HDFS への変更をログ先行書き込み方式で記録する。これをジャーナルと呼び、ネイティブファイルシステム上につくられる。ブロックのレプリカは、永続チェックポイントとは別の場所にある。

クライアント主導で始まったトランザクションはジャーナルに記録され、ジャーナルファイルのフラッシュと同期が済んでからクライアントへの通知を送信する。チェックポイント

[2] http://www.lustre.org

ファイルは、決して NameNode から書き換えられることはない。新しいファイルが書かれるのは、再起動中にチェックポイントが作られたり管理者から要求があったり、あるいは次のセクションで紹介する CheckpointNode からの要求があったりしたときだ。スタートアップ時に、NameNode はチェックポイントから名前空間イメージを立ち上げる。そしてジャーナルの変更を再生していく。新たなチェックポイントと空のジャーナルをストレージのディレクトリに書き出したら、NameNode はクライアントに対応できるようになる。

永続性を向上させるため、チェックポイントやジャーナルの冗長なコピーをとっておくことが一般的だ。複数の独立したローカルボリュームやリモートの NFS サーバーなどにコピーを格納する。複数のローカルボリュームに置いておけばどれかひとつのボリュームに障害が発生してもデータを失わずに済むし、リモートの NFS サーバーに置いておけばノード全体の障害からもデータを守れる。NameNode がジャーナルをストレージに書き込む際にエラーが発生すると、自動的にそのストレージをストレージディレクトリのリストから除外するようになる。また、使えるストレージディレクトリがひとつもなくなった時点で NameNode は自動的に自分自身をシャットダウンさせる。

NameNode はマルチスレッドに対応しており、複数のクライアントからのリクエストを同時に処理する。そのため、トランザクションのディスクへの保存がボトルネックとなる。どれかひとつのスレッドがフラッシュからシンクにいたる手続きを開始すると、それが完了するまで他のスレッドが待つ必要があるからである。この流れを最適化するため、NameNode は複数のトランザクションを一括で処理する。どれかひとつの NameNode のスレッドがフラッシュとシンクの操作を始めると、その時点でコミット済みのすべてのトランザクションをひとまとめにして処理する。残りのスレッドは、自分のトランザクションが保存されたかどうかだけを確認すればよいのであって、個別にフラッシュとシンクをする必要はない。

DataNodes

DataNode 上の各ブロックのレプリカは、ネイティブファイルシステム上では二つのファイルで表される。ひとつはデータそのものを含むファイルで、もうひとつはブロックのメタデータを含むファイルだ。メタデータの中には、データのチェックサムやタイムスタンプが格納されている。データファイルのサイズは実際のブロックの長さと等しくなる。昔ながらのファイルシステムみたいに、規定のブロックサイズにそろえるために余分なスペースが必要などということはない。したがって、ブロックの中身が半分空っぽだった場合は、ローカルドライブ上で必要となる容量も半分で済む。

各 DataNode は、開始時に NameNode に接続してハンドシェイクを行う。ハンドシェイクの目的は、名前空間 ID や DataNode のソフトウェアのバージョンを検証することだ。そのいずれかが NameNode のものと一致しない場合、DataNode は自動的に終了する。

名前空間 ID は、ファイルシステムのインスタンスに割り当てられる。また、名前空間 ID はクラスタの全ノードに永続的に格納される。異なる名前空間 ID のノードを同じクラスタにまとめることはできないようにして、ファイルシステムの整合性を守っている。新しく作っ

たDataNodeでまだ名前空間IDがないものはクラスタに組み込むことができ、組み込んだクラスタの名前空間IDを受け取る。

ハンドシェイクを終えると、DataNodeをNameNodeに登録する。DataNodesは、自身の一意なストレージIDを永続的に保存する。ストレージIDはDataNodeの内部IDで、別のIPアドレスやポートで立ち上げなおしたときにも認識できる。ストレージIDがDataNodeに割り振られるのは最初にNameNodeに登録したときであり、それ以降は決して変わらない。

DataNodeは、NameNodeの持つブロックレプリカを識別するために、ブロックレポートを送信する。ブロックレポートの内容は、ブロックIDやタイムスタンプと、サーバーが保持する各ブロックレプリカの長さである。最初のブロックレポートは、DataNodeを登録した直後に送信される。それ以降のブロックレポートは一時間ごとに送信され、NameNodeの情報と、ブロックレプリカがクラスタ上のどこにあるのかの最新情報を提供する。

通常の操作中は、DataNodesからNameNodeにハートビートを送信する。これによって、DataNodeが動作中であることやそこで保持するブロックレプリカが利用可能であることを確認する。デフォルトのハートビート間隔は3秒である。NameNodeがDataNodeからのハートビートを10分以上受信できなかった場合、NameNodeはそのDataNodeがダウンしていると判断する。そして、そのDataNodeが管理するブロックレプリカにはアクセスできないようにする。その後、そのブロックを他のDataNode上で扱うための新たなレプリカの作成スケジュールを入れる。

DataNodeからのハートビートでは、それ以外にもさまざまな情報を運ぶ。ストレージの総容量や実際に使っているストレージの割合、そして現在進行中のデータ転送の数などである。これらの統計情報をもとに、NameNodeのブロック配置やロードバランシングに関する決定を下す。

NameNodeからDataNodeに直接リクエストを送ることはない。ハートビートへの応答で、DataNodeへの指示を送信する。この指示に含まれるのは、他のノードへのブロックの複製やローカルブロックレプリカの削除、ブロックの再登録と最新状態の報告、ノードのシャットダウンといったコマンドである。

これらのコマンドは、システム全体の整合性を保つために重要となる。従って、大規模なクラスタであってもハートビートは頻繁に送る必要がある。NameNodeは、秒間数千回ものハートビートを処理してもその他の操作に影響を及ぼさないようになっている。

HDFSクライアント

ユーザーアプリケーションからファイルシステムにアクセスするときに使うのがHDFSクライアントである。これは、HDFSのファイルシステムのインターフェイスを公開したライブラリだ。

既存の大半のファイルシステムと同様、HDFSでもファイルの読み書きや削除をサポートしている。また、ディレクトリを作ったり削除したりすることもできる。利用者がファイルやディレクトリを指すときには、名前空間内でのパスを使う。アプリケーション側では、ファ

イルシステムのメタデータやストレージがどのサーバーにあるかは知らなくてもよいし、そのブロックが複数のレプリカを持っているかどうかなども知る必要はない。

アプリケーションからファイルを読み込むときに、HDFS クライアントはまず最初に NameNode に問い合わせる。そのファイルのブロックのレプリカを持つ DataNodes のリストを得るためである。このリストは、クライアント側から見たネットワークトポロジー上の距離の順に並べ替えられる。クライアントは DataNode に直接接触し、必要なブロックの転送をリクエストする。クライアントがファイルを書き込むときは、まず NameNode に問い合わせて、そのファイルの最初のブロックのレプリカを持つ DataNode を調べる。次にクライアントがノード間のパイプラインを構成し、データを送信する。最初のブロックがいっぱいになれば、次のブロックのレプリカを管理する新しい DataNode をクライアントがリクエストする。新たなパイプラインが作られ、そしてクライアントがファイルの続きのデータを送信する。各ブロックで選択する DataNode はさまざまになる。クライアントと NameNode そして DataNodes の間のやりとりを図 9.1 に示す。

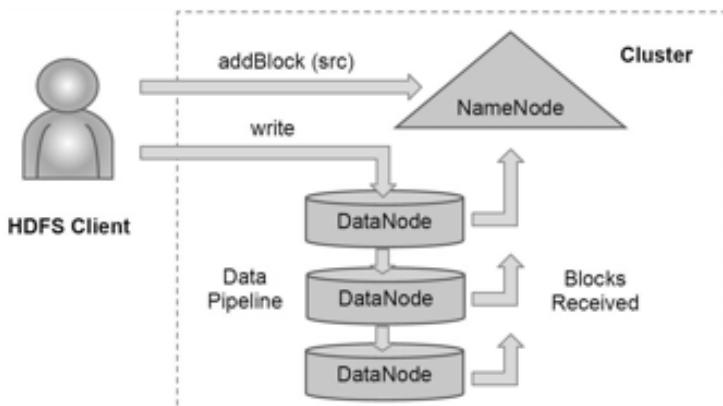



図 9.1: HDFS クライアントによる新たなファイルの作成

既存のファイルシステムとは異なる点として、HDFS にはファイルブロックの場所を公開する API が用意されている。これを使えば、MapReduce フレームワークなどのアプリケーションがデータをどこに配置するかをスケジュールできるようになり、結果として読み込みのパフォーマンスを向上させることができる。また、アプリケーション側でファイルのレプリケーション係数を決めることもできる。デフォルトでは、ひとつのファイルのレプリケーション係数は 3 である。クリティカルなファイルや頻繁にアクセスされるファイルの場合は、レプリケーション係数を大きめにしておけば耐障害性が高まるし読み込みの帯域も大きくできる。

CheckpointNode

HDFS における NameNode の主要な役割はクライアントからのリクエストに対応することだ。それ以外にも、他の役割である CheckpointNode あるいは BackupNode を演じることもできる。どんな役割を担当するかは、ノードの開始時に指定する。

CheckpointNode は、既存のチェックポイントとジャーナルを定期的に結合し、新しいチェックポイントと空のジャーナルを作る。CheckpointNode は、通常は NameNode とは別のノードで稼働する。というのも、NameNode と同程度のメモリを必要とするからである。CheckpointNode は現時点のチェックポイントとジャーナルファイルを NameNode からダウンロードし、ローカルでそれをマージして、新たなチェックポイントを NameNode に返す。

定期的なチェックポイントの作成は、ファイルシステムのメタデータを守るひとつの方法でもある。仮に名前空間イメージやジャーナルの永続コピーがすべて使えなくなったとしても、直近のチェックポイントからシステムを再開させることができる。チェックポイントを作れば、新たなチェックポイントを NameNode にアップロードした時点でジャーナルを切り詰めさせることもできる。HDFS クラスタを再起動なしで長期間稼働させると、ジャーナルがどんどん大きくなっていく。サイズが巨大になると、ジャーナルファイルを失ったり壊してしまったりする可能性も増える。また、ジャーナルが巨大になると NameNode の再起動にも時間がかかるようになる。大規模なクラスタでは、一週間のジャーナルを処理するのに一時間ほどかかる。おすすめは、毎日チェックポイントを作っておくことだ。

BackupNode

HDFS に最近導入された機能が BackupNode だ。CheckpointNode と同様に、BackupNode にも定期的なチェックポイント作成機能がある。しかしそれだけでなく、インメモリで最新のファイルシステム名前空間のイメージを保持できる。これは、NameNode の現状と常に同期したものとなる。

BackupNode は、名前空間のトランザクションのジャーナルストリームをアクティブな NameNode から受け取り、それを自前のストレージディレクトリ上のジャーナルに保存する。そして、そのトランザクションを、自身が持つインメモリの名前空間イメージに適用する。NameNode は BackupNode をジャーナルの保存場所とみなし、自分のストレージディレクトリにあるジャーナルファイルと同様に扱う。NameNode がダウンしたときは、BackupNode がインメモリに持つイメージとディスク上のチェックポイントが最新状態の記録となる。

BackupNode は、アクティブな NameNode からチェックポイントやジャーナルファイルをダウンロードしなくてもチェックポイントを作ることができる。というのも、既に最新の名前空間イメージをメモリ上に保持しているからである。そのおかげで BackupNode でのチェックポイント処理がより効率的に行える。名前空間の内容をローカルストレージディレクトリに保存するだけで済むからである。

BackupNode は、読み込み専用の NameNode として見ることもできる。ブロックの場所を除いたすべてのファイルシステムメタデータの情報を持っているのだ。名前空間の変更をする操作やブロックの場所がわからないとできない操作を除いて、通常の NameNode でできる操作なら何でも実行できる。BackupNode を使えば、永続ストレージなしで NameNode を動かすという選択肢も可能となる。名前空間の状態を永続化させる役割を BackupNode に委譲するというわけだ。

アップグレードおよびファイルシステムスナップショット

ソフトウェアのアップグレードのときには、バグや操作ミスなどでファイルシステムが壊れてしまう可能性が高まる。HDFS でスナップショットをつくる目的は、アップグレードの際にシステムに格納されたデータに与えるダメージを最小限に抑えることだ。

スナップショットを使えば、管理者はファイルシステムの現状を永続的に保存できる。もしアップグレードしたせいでデータが消えてしまったり壊れてしまったりしても、アップグレードを取り消して名前空間とストレージをスナップショット作成時の状態に戻すことができる。

スナップショットはただ一つだけ保持することができるもので、システムが立ち上がるときにクラスタ管理者のオプションで作られる。スナップショットを作るようリクエストを受けると、NameNode はまずチェックポイントとジャーナルファイルを読み込んで、メモリ内でそれをマージする。それから、新たなチェックポイントと空のジャーナルを新しい場所に書き込む。古いチェックポイントとジャーナルを変更せずに済ませるためだ。

ハンドシェイクの際に、ローカルスナップショットを作るかどうかの指示を NameNode から DataNodes に出す。DataNode 上のローカルスナップショットは、データファイルを含むディレクトリを複製して作るわけにはいかない。そんなことをすると、クラスタ上のすべての DataNode に通常の倍のストレージ容量が必要となってしまうからだ。そのかわりに、各 DataNode はストレージディレクトリだけをコピーして、既存のブロックファイルのハードリンクをそこに作成する。DataNode がブロックを削除するときにはハードリンクだけを削除し、追記などでブロックを変更するときにはコピー・オン・ライト方式を使う。これで、古いブロックのレプリカはそのまま旧ディレクトリに残ることになる。

クラスタ管理者は、システムを再起動するときに HDFS をスナップショットの状態に戻すことができる。このとき、NameNode はスナップショット作成時に保存したチェックポイントを復元する。DataNode は以前にリネームしたディレクトリを復元し、バックグラウンドプロセスを立ち上げて、スナップショットを作成した後にできたブロックレプリカを削除する。ロールバックを選択すると、ロールフォワードはできなくなる。クラスタ管理者は、スナップショットが占有していたストレージを解放させるためにスナップショットの破棄を指示できる。アップグレードの際に作ったスナップショットについては、ここまですればアップグレード完了となる。

システムが進化するにつれて、NameNode のチェックポイントやジャーナルファイルのフォーマットが変わる可能性がある。あるいは、DataNode におけるブロックレプリカファイルでのデータの表現方法も変わるかもしれない。データの表現フォーマットを示すのがレイアウトバージョンで、これは NameNode や DataNode のストレージディレクトリに格納されている。各ノードは起動時に、現在のソフトウェアのレイアウトバージョンと自分のストレージディレクトリのレイアウトバージョンを比較する。そして、旧バージョンのフォーマットは自動的に新バージョンに変換する。この自動変換をするには、新しいレイアウトバージョンのソフトウェアにアップグレードしてシステムを立ち上げなおすときにスナップショットを作ることが必須となる。

9.3　ファイル I/O 操作およびレプリカの管理

当然ながら、ファイルシステムはファイルのデータを格納できてナンボのものだ。それを HDFS でどう実現しているのかを理解するには、読み書きをどのように行っているのか、そしてブロックをどのように管理しているのかを知る必要がある。

ファイルの読み書き

アプリケーションから HDFS にデータを追加するときには、新しいファイルを作ってデータをそこに書き込む。いったんファイルをクローズしたら、書き込んだデータを変更したり削除したりすることはできない。ただし、新たなデータを追加することはできる。この場合は、ファイルを追記モードで再オープンする。HDFS で実装しているのは、シングルライター・マルチリーダーモデルである。

HDFS クライアントがファイルを書き込み用にオープンすると、そのファイルへの書き込み権限がリースされる。その間、他のクライアントは同じファイルに書き込めない。書き込み側のクライアントは、ハートビートを NameNode に送信することによって定期的にリースを更新する。ファイルをクローズするときに、リースが破棄される。リース期間は、ソフトリミットとハードリミットによる制約を受ける。ソフトリミットに達するまでは、ライターはそのファイルへの排他アクセス権があるものと確信している。ソフトリミットを超えてからクライアントがファイルのクローズやリースの更新に失敗すると、別のクライアントがそのリースを横取りできるようになる。ハードリミット (1 時間) を超えてからクライアントがファイルのクローズやリースの更新に失敗すると、HDFS はそのクライアントが終了したものとみなす。そのときはライターがファイルを自動的にクローズし、リースを解放する。ライターがリースを確保している間も、他のクライアントからのファイルの読み込みは可能である。そのため、ひとつのファイルを同時に複数のリーダーが読むこともあり得る。

HDFS のファイルは複数のブロックで構成されている。新しいブロックが必要になると、NameNode は新しいブロックを確保して一意なブロック ID を割り当て、そのブロックのレプ

リカを持つ DataNode のリストを決める。DataNode はパイプラインを形成しており、その並び順は、クライアントから最後の DataNode までのネットワーク上の距離が最短になるようにする。バイトデータをパイプラインに流すときには、一連のパケット群として流す。アプリケーションから書き込むバイトデータは、まずクライアント側でバッファリングする。パケットバッファ(通常は 64KB) がいっぱいになると、データをパイプラインに送り出す。その次のパケットをパイプラインに送り出すのは、前のパケットの受領通知を受け取った後になる。未処理のパケットの最大数は、クライアントの未処理パケットウィンドウのサイズまでに制限される。

データが HDFS のファイルに書き込まれても、そのファイルをクローズするまではリーダーから見えることが保証されない。アプリケーション側で見えるようになることを保証したければ、hflush を明示的に呼べばよい。そうすれば、現在のパケットが即時にパイプラインに送られる。そして、パイプライン内のすべての DataNode がパケットの送信に成功したことを確認するまで待ち続ける。これで、hflush より前に書き込まれたすべてのデータは確実にリーダーから見えるようになる。

何もエラーが発生しなければ、ブロックの作成は 3 つのステージを経て行われる。図 9.2 にその様子を示した。3 つの DataNode(DN) と 5 つのパケットからなるブロックのパイプラインである。図中の太線で示されているのがデータパケット、破線は確認メッセージ、そして細線はパイプラインの準備や終了用の制御メッセージとなる。縦線がクライアントと各 DataNode で発生するアクティビティを表し、時間の経過とともに上から下へと流れていく。`t0` から `t1` までが、パイプラインの準備ステージだ。`t1` から `t2` までがデータストリーミングステージで、`t1` が最初のデータパケットの送信時刻、そして `t2` が最終パケットの受信確認時刻となる。またこのとき、hflush 命令を `packet 2` で送っている。hflush の指示はパケットデータとともに流れ、個別の命令とはならない。最後の `t2` から `t3` までが、このブロックのパイプライン終了ステージである。

何千ものノードからなるクラスタでは、どれかひとつのノードに障害が発生する (たいていはストレージ障害) なんてことは日常茶飯事だ。DataNode に格納されているレプリカは、メモリやディスクあるいはネットワークの障害などのせいで壊れる可能性がある。HDFS では、HDFS ファイルの各データブロックのチェックサムを計算して保存する。HDFS クライアントがチェックサムを検証し、クライアントや DataNode あるいはネットワークの障害によるデータの破壊を発見しやすくする。クライアントが HDFS ファイルを作るときには、各ブロックのチェックサムを計算して、それをデータとともに DataNode に送信する。DataNode ではチェックサムをメタデータファイルに保存して、ブロックのデータファイルとは別にする。HDFS がファイルを読み込むときには、各ブロックのデータとチェックサムをクライアントに送る。クライアントは受け取ったデータのチェックサムを計算し、その結果が実際に受け取ったチェックサムと一致することを確かめる。もし一致しなければ、そのレプリカの NameNode を通知する。そして、同じブロックの別のレプリカを別の DataNode から取得する。

読み込みたいファイルをクライアントがオープンするときには、ブロックのリストを取得して、各ブロックのレプリカの場所を NameNode から取得する。各ブロックの場所は、リー

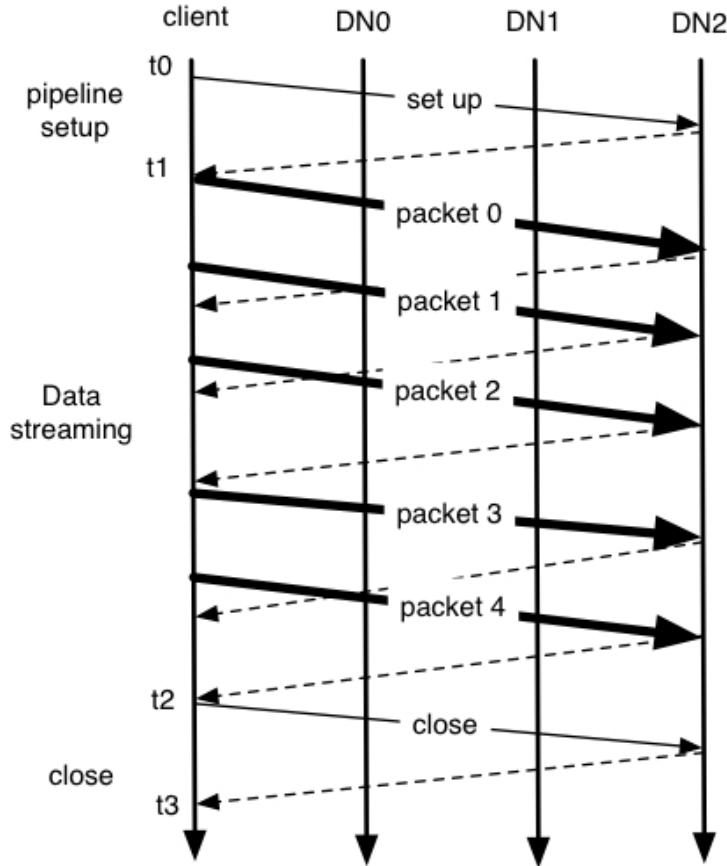



図 9.2: ブロックを書き込むときのデータパイプライン

ダーからの距離の順になっている。ブロックの内容を読むときには、一番近いレプリカから順に読もうと試みる。読み込めなければ、リスト上でその次にあるレプリカからの読み込みを試みる。読み込みが失敗するのは、DataNode に到達できなかったりそのノードが対象ブロックのレプリカを持っていなかったり見つかったレプリカがチェックサムの検証に失敗したりといった場合だ。

HDFS では、書き込み用にオープンされているファイルでも別のクライアントから読み込める。書き込み用にオープンされているファイルを読み込みときは、現在書き込み中の最後のブロックの長さは NameNode からはわからない。そんな場合は、クライアントがいずれかのレプリカに最新の長さを問い合わせてからその内容を読み始める。

HDFS の I/O の設計は MapReduce のようなバッチ処理システムに最適化されており、シーケンシャルリード/ライトについては高いスループットを要求される。リアルタイムのデータ

ストリーミングやランダムアクセスを必要とするアプリケーション向けに、読み書きのレスポンスタイムを改善するための作業は今でも続いている。

ブロックの配置

大規模なクラスタでは、すべてのノードをフラットなトポロジーでつなげるのは非現実的だ。よくある方法は、複数のラックにまたがってノードを展開する方法である。ひとつのラック上のノード群はスイッチを共有し、ラックのスイッチがひとつあるいは複数のコアスイッチと接続される。異なるラック上にあるふたつのノード間の通信は、複数のスイッチを経由する必要がある。多くの場合、同一ラック内のノード間のほうが異なるラックの場合よりもネットワーク帯域が広い。図 9.3 はふたつのラックからなるクラスタの例で、それぞれのラックに 3 ノードずつ含まれている。

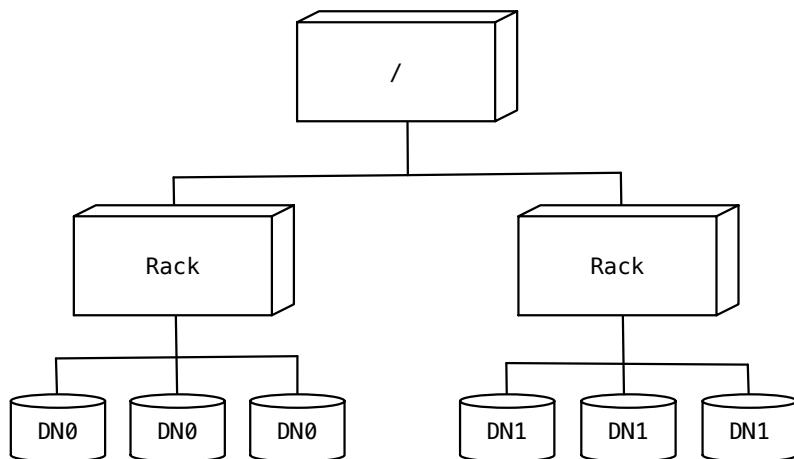



図 9.3: クラスタのトポロジー

HDFS は、ふたつのノード間のネットワーク帯域の見積もりを、両者の距離に基づいて行う。あるノードからその親ノードへまでの距離を 1 として、ふたつのノードの距離は一番近い共通の祖先からの距離の合計で求める。ノード間の距離が短いほど、データの転送により多くの帯域を使えることを意味する。

HDFS の管理者は、ノードのアドレスを指定すればそのノードのラック情報を返すようなスクリプトを用意できる。NameNode が中心となって、各 DataNode のラックの場所を解決する。DataNode を NameNode に登録するときに、NameNode がそのスクリプトを実行し、ノードが所属するラックを決める。スクリプトを準備していない場合は、NameNode はすべてのノードがデフォルトの単一ラックに属しているものとみなす。

レプリカをどこに配置するかは、HDFS のデータの信頼性や読み書きのパフォーマンスを考えると重要である。配置ポリシーをうまく指定できればデータの信頼性や可用性があがる

し、ネットワーク帯域もうまく活用できる。現在 HDFS は、ブロック配置ポリシー設定用のインターフェイスを提供している。これを使えば、ユーザーや研究者がいろいろなポリシーを試して自分たちのアプリケーションに最適なポリシーを設定できるようになる。

デフォルトの HDFS ブロック配置ポリシーは、書き込みのコストを最小化することとデータの信頼性や可用性の最大化や読み込み帯域の集約とのトレードオフとなる。新しいブロックを作成する場合、HDFS はノードの最初のレプリカをライターのある場所に配置する。第二、第三のレプリカは、別のラックにあるそれぞれ別のノードに配置する。残りのレプリカはランダムなノードに配置する。ただし、同一ノード上に複数のレプリカは配置しないし、同一ラック上にも最大で二つまでのレプリカしか配置しないようにする。第二、第三のレプリカを別のラックに配置することで、あるひとつのファイルのレプリカがクラスタをまたがってうまく分散させられるようになる。もし最初の二つのレプリカが同じラックに入ってしまうと、どのファイルに対してもブロックレプリカの 2/3 が同じラックにあることになってしまう。

すべてのターゲットノードが選べたら、ノード群をパイプライン化する。その順番は、最初のレプリカに近い順となる。データはこの順でノードに送られていく。読み込みの場合、NameNode はまずクライアントのホストがそのクラスタ内にあるかどうかを調べる。ある場合は、ブロックの場所をクライアントに返す。その順番は、リーダーに近い順となる。この順に従って、ブロックを DataNode から読み込む。

このポリシーに従えば、ラック間やノード間の書き込みトラフィックを軽減でき、一般に書き込みのパフォーマンスが向上する。ラックに障害が発生する可能性はノードの障害よりもはるかに低いので、このポリシーはデータの信頼性や可用性にはあまり影響を及ぼさない。三つのレプリカを持つ通常の場合、このポリシーを使えばデータ読み込み時のネットワーク帯域を集約できる。というのも、あるブロックのある場所がどこか二つのラックに絞れるからである。

レプリケーションの管理

NameNode は、各ブロックが常に十分な数のレプリカを確保できているように努める。あるブロックのレプリカが少なすぎたり多すぎたりしないかどうかを、NameNode は DataNode からのブロックレポートで判断する。あるブロックのレプリカが多すぎる場合、NameNode はどのレプリカを削除するかを選択する。レプリカをホストしているラックの数をできるだけ減らさないことを第一の前提とし、そのうえでディスクの残容量がいちばん少ない DataNode のレプリカから削除していく。目標は、DataNode 間でのストレージ使用料のバランスをとりつつブロックの可用性を落とさないようにすることだ。

ブロックが少なくなりすぎると、レプリケーションの優先キューに投入される。レプリカがひとつしかないブロックが最優先であり、レプリケーション係数の 2/3 より多いレプリカがあるブロックは最も優先度が低くなる。バックグラウンドスレッドが定期的にレプリケーションキューの先頭をスキャンし、新たなレプリカを作るかどうかを決める。ブロックのレプ

リカを作るときには、新たなブロックを配置するときと同様のポリシーに従う。もし既存のレプリカの数がひとつだけなら、HDFS は次のレプリカを別のラック上に作る。既存のレプリカがふたつある場合は、もしそれらが同じラック上にあれば三番目のレプリカを別のラック上に配置する。そうでない場合は、三番目のレプリカを既存のレプリカと同一ラック上の別のノードに配置する。ここでの目標は、新たなレプリカの作成コストを削減することだ。

また、NameNode は、全レプリカが同一ラック上に配置されることがないようにもする。あるブロックのレプリカがすべて同じラック上にあることを検出すると、NameNode はそのブロックを「レプリケーションできていない」とみなし、先ほどと同様のポリシーに基づいて別のラック上にレプリカを作成する。レプリカができたという通知を NameNode が受け取ったら、そのブロックはレプリカが多すぎる状態になる。そこで、NameNode は古いレプリカのひとつを削除する。多すぎるレプリカへの対応ポリシーに基づいて、ラック数が減らないようにするためである。

バランサー

HDFS のブロック配置戦略では、DataNode のディスク利用状況は考慮しない。これは、新しいデータ—つまり、より参照されやすいデータ—を大量の空き領域がある DataNode のごく一部に配置されてしまわないようにするためだ。そのため、データが各 DataNode にまんべんなく配置されるとは限らない。また、新たなノードがクラスタに追加されたときにも不均衡が生じる。

バランサーは、HDFS クラスタ上のディスク使用量のバランスをとるためのツールである。0 から 1 までの間の小数で表した閾値を、入力として受け取る。クラスタのバランスが取れているとみなすのは、クラスタ内の各 DataNode について、そのノードでの利用率[3] とクラスタ全体での利用率[4] との差が閾値を超えないときである。

このツールはアプリケーションプログラムとして配布されており、クラスタの管理者が実行することができる。実行すると、使用率の高い DataNode から使用率の低い DataNode へとレプリカを移動させる。バランサーでポイントとなる要件のひとつは、データの可用性を保つことである。移動させたいレプリカを選んでその移動先を見つけるときにも、それによってレプリカ数やラック数が減らないことを保証する。

バランサーは、その処理を最適化するために、ラックをまたがるデータコピーを最小限に抑える。レプリカ A を別のラックに移動させる必要があると判断したときに、移動先のラックにもし同じブロックのレプリカ B が存在すれば、レプリカ A ではなくレプリカ B からデータをコピーする。

設定パラメータで、バランシング操作で使う帯域を制限できる。利用できる帯域を高めに設定すると、バランスのとれた状態に素早く持ち込める。しかしそのぶん、アプリケーションの処理に影響が及んでしまう。

[3] ノードの全容量に対する利用中の容量の比率。
[4] クラスタの総容量に対する利用中の容量の比率。

ブロックスキャナー

各 DataNode ではブロックスキャナーが動作している。これは、定期的にブロックのレプリカをスキャンし、保存されているチェックサムがブロックのデータと一致するかを調べる。スキャン時には読み込みの帯域を調節し、設定した期間で検証を終えられるようにする。あるクライアントがどれかのブロック全体を読み込んでチェックサムの検証に成功すると、それが DataNode に伝えられる。そして DataNode では、そのレプリカが検証されたものとみなす。

各ブロックをいつ検証したのかという情報は、可読形式でログファイルに記録される。DataNode のトップレベルディレクトリに、現在のログと前回のログのふたつのファイルが存在する。新たに検証をすると、その時刻が現在のログファイルに追記される。それに対応して、各 DataNode にはインメモリのスキャンリストがあり、これはレプリカの検証時刻順で並んでいる。

クライアントからの読み込みやブロックスキャナーのスキャンでブロックの破損が見つかると、それが NameNode に通知される。NameNode はそのレプリカが壊れているとマークするが、すぐにはそのレプリカを削除しない。その代わりに、まずそのブロックの壊れていないコピーを複製する。正常なレプリカの数がブロックのレプリケーション係数に達してから、壊れたレプリカの削除をする。このポリシーは、データをできるだけ長く永続させることを狙ったものである。仮にすべてのレプリカが壊れていたとしても、このポリシーにしておけば、壊れたレプリカからユーザーがデータを取得できるようになる。

廃止措置

クラスタ管理者が、廃止予定のノードのリストを指定する。ある DataNode が廃止予定とマークされると、もうそのノードはレプリカの配置先として選ばれなくなる。しかし、読み込みリクエストへの対応は続行する。そして NameNode は、廃止予定のノードにあるブロックのレプリカを他の DataNode に移し始める。すべてのブロックが他の DataNode にレプリケートできたことを確認した時点で、そのノードは廃止状態となる。廃止状態のノードは、データの可用性に一切影響を及ぼさず安全に削除できる。

クラスタ間データコピー

大規模なデータセットでは、HDFS クラスタへのデータのコピー（あるいは HDFS クラスタからのデータのコピー）は手強い作業である。HDFS には DistCp というツールが用意されており、クラスタ内/クラスタ間での大規模な並列コピーに使える。これは MapReduce ジョブである。個々のマップタスクが、ソースデータの一部をコピー先のファイルシステムにコピーする。MapReduce フレームワークが、並列タスクスケジューリングやエラーの検出そしてリカバリーを自動的に処理する。

9.4　Yahoo!における実例

Yahoo!の大規模 HDFS クラスタは、約 4000 ノードで構成されている。標準的なクラスタノードのハードウェアは、2.5 GHz のクアッドコア Xeon プロセッサを 2 台搭載し、4–12 本の SATA ドライブを直結 (各 2TB)、RAM は 24GB、そして 1 ギガビットの Ethernet 接続がある。ディスク領域の 7 割が HDFS 用に割り当てられており、残りの領域は、OS(Red Hat Linux) やログ、そして Map タスクでの出力用の領域である (MapReduce の中間データは、HDFS に格納しない)。

ひとつのラックに 40 ノードが格納されており、これらがひとつの IP スイッチを共有する。ラックのスイッチは、8 台あるコアスイッチのいずれかと接続される。コアスイッチは、ラック群をクラスタ外のリソースと接続できるようにするものである。それぞれのクラスタで、NameNode と BackupNode のホストは特別な設定になっており、最大で 64GB の RAM を積んでいる。アプリケーションのタスクがこれらのホストに割り振られることは決してない。全体的に見ると、4000 ノードのクラスタでブロックとして使えるストレージは 11PB(ペタバイト; 1000TB) である。三重にレプリケートされているので、アプリケーションから使えるストレージは 3.7PB ということになる。長年 HDFS を使ってきて、クラスタノード用に選択するホストも技術の進歩の恩恵を受けている。新しいクラスタノードはいつもこれまでよりも高速なプロセッサを搭載しているし、ディスクも大きければ RAM も多い。昔からある低速で少容量のノードは引退させることもあるし、Hadoop の開発やテスト用のクラスタに転用することもある。

例として取り上げた大規模クラスタ (4000 ノード) で扱っているのは、約 6500 万のファイルと 8000 万のブロックである。各ブロックは一般に三度レプリケートされるので、すべてのデータノードが約 60,000 ブロックのレプリカを保持することになる。ユーザーアプリケーションは、毎日約 200 万のファイルをクラスタ上に新しく作る。Yahoo!の Hadoop クラスタにある 40,000 ノードが、オンラインデータストレージとして 40PB を提供する。

Yahoo!のテクノロジー群の中で重要な位置を占めるようになるということは、さまざまな技術的問題に取り組むことを意味する。研究プロジェクトのひとつであることとペタバイト級の業務データを管理することには違いがあるのだ。真っ先に課題となるのが、データの安定性と永続性である。しかしそれ以外にも重要なことがある。コストパフォーマンス、ユーザーコミュニティのメンバー間でのリソース共有の仕組み、そしてシステムの運用担当者による管理のしやすさなどである。

データの永続性

データを三重にレプリケーションすることで、相関性のないノード障害によるデータのロストに対する強力な守りとなる。Yahoo!では、今までこのパターンでデータを失ったことはない。大規模なクラスタで、1 年の間にひとつのブロックを失う確率は 0.005 未満だ。知っておくべきなのは、毎月約 0.8 パーセントのノードに障害が発生するということだ (そのノード

を最終的にリカバーすることになっても、そこで管理していたデータの復旧に手間をかけることはない)。つまり、先ほど例に出した大規模クラスタだと、毎日ひとつかふたつのノードを失うことになる。このクラスタでは、障害が発生したノード上にあった 60,000 ブロックレプリカの再作成を約 2 分で行う。再レプリケーションが高速に行えるのは、それがクラスタのサイズに対応する問題だからである。2 分の間に複数のノードで同時に障害が発生し、どれかひとつのブロックの全レプリカが失われるという確率はごく小さい。

相関性のあるノードの障害は、また別の脅威となる。この種の障害で最もよくあるのが、ラックあるいはコアスイッチの障害である。HDFS はラックスイッチを失っても耐えられるようになっている (各ブロックのレプリカを、どこか別のラック上に保持している)。コアスイッチの障害は、複数のラックにまたがるクラスタの一部を事実上切り離してしまうことになる。そんな場合はいくつかのブロックが利用できなくなる可能性がある。いずれにせよ、スイッチを復旧させれば利用できなかったレプリカをクラスタに復元できる。それ以外に関連のある障害としては、クラスタへの電源の停電 (アクシデントによるものも計画的なものも含む) がある。複数のラックが停電すると、おそらくいくつかのブロックは使えなくなるだろう。しかし、電源が復旧したからといってそれで元通りになるとは限らない。0.5 パーセントから 1 パーセントのノードは、電源オンからの起動でうまく立ち上がらない。統計的でも、実際の経験上でも、大規模なクラスタで電源オンからやり直したときにはごく少数のブロックを失ってしまうことが知られている。

ノード自体の障害に加えて、ノードに格納されているデータも破壊されたり失ったりすることがある。ブロックスキャナーが大規模クラスタの全ブロックを二週間おきにスキャンし、毎回 20 前後の不良レプリカを見つけている。不良レプリカは、検出された時点でリプレイスされる。

HDFS の共有機能

HDFS の利用が増えるにつれて、ファイルシステム自体にもリソースの共有手段を導入する必要が出てきた。さまざまな数多くのユーザー間での共有だ。最初に用意された機能はパーミッションフレームワークで、これは Unix のファイルやディレクトリのパーミッションと似た仕組みだった。このフレームワークでは、ファイルやディレクトリにそれぞれ個別のアクセス権を設定する。所有者用のアクセス権、ファイルやディレクトリに関連付けられたユーザーグループ用のアクセス権、そして他のすべてのユーザー用のアクセス権である。Unix (POSIX) のパーミッションと HDFS のパーミッションの違いは、HDFS における通常のファイルには実行権限やスティッキービットが存在しないという点だ。

初期のバージョンの HDFS では、ユーザーの身元確認が弱点だった。アクセス元のホストが主張する身元情報をそのまま受け入れるしかできなかったのだ。HDFS にアクセスするときに、クライアントアプリケーションがローカルの OS に対してユーザーやグループの情報を問い合わせていた。新しいフレームワークの場合、クライアントアプリケーションは信頼できるソースから取得した認証情報を使う必要がある。認証管理にはさまざまな方式を利用で

き、初期状態の実装では Kerberos を使っている。ユーザーアプリケーションは、同じフレームワークを使えば、信頼できる認証情報を持っているかどうかを確認できる。また、クラスタに参加する各データノードへの認証を要求することもできる。

データストレージとして使える容量の総計は、データノードの数と各ノードに用意されたストレージで決まる。HDFS を使い始めたころの経験からわかったのが、ユーザーコミュニティをまたがるリソース配置ポリシーを何らかの手段で設定できるようにする必要性だった。公平に共有させることも大切だが、それだけではない。ユーザーのアプリケーションが何千ものホストにデータを書き込む場合、そのアプリケーションがリソースを食いつぶしてしまわないようにすることも重要だ。HDFS の場合、システムのメタデータが常に RAM 上に存在するので、名前空間のサイズ (ファイルやディレクトリの数) もまた有限なリソースとなる。ストレージや名前空間のリソースを管理するために、各ディレクトリに容量制限を課すこともできる。そのディレクトリ配下の名前空間のサブツリーでファイルを保存するための総容量で制限する。それ以外にも、サブツリー内に格納できるファイルやディレクトリの総数を制限することもできる。

HDFS のアーキテクチャは、大半のアプリケーションが入力として大規模なデータセットを流し込むということを前提としている。一方、MapReduce プログラミングフレームワークには、小さめのファイルを大量に (Reduce タスクの数と同じだけ) 生成するという傾向がある。これは、名前空間のリソースにとってはさらなるストレスとなる。利便性を考慮して、ディレクトリのサブツリーを単一の Hadoop Archive(HAR) ファイルにまとめることもできる。HAR ファイルは、おなじみの tar や JAR あるいは Zip ファイルと似たようなものだが、ファイルシステム上の操作でアーカイブ内の個別のファイルを扱える。また、HAR ファイルを MapReduce ジョブへの入力として透過的に使うこともできる。

スケーリングおよび **HDFS Federation**

NameNode のスケーラビリティは、これまでずっと課題であった [Shv10]。NameNode はすべての名前空間やブロックの場所をメモリに保持するので、NameNode のヒープサイズのせいで扱えるファイル数に制約が出てしまい、アドレス可能なブロック数にも制約が出てしまう。同じ理由で、NameNode がサポートできるクラスタストレージの総容量も限られる。ユーザーは、より大きなファイルを作るよう求められる。しかし、そのためにはアプリケーションの振る舞いを変更しないといけないので、なかなかそうはならない。さらに、最近の HDFS 用アプリケーションの中には、小さいファイルを大量に格納しないといけないものも出てきている。使用量を管理するために制限機能が追加されたし、アーカイブツールも用意した。しかしこれらを使っても、スケーラビリティの問題への本質的な対策にはならない。

新機能として、複数の独立した名前空間 (および NameNodes) でクラスタ内の物理ストレージを共有できるようにした。名前空間が、ブロックプールの配下にまとめられたブロックを使うようにするものだ。ブロックプールとは SAN ストレージシステムにおける論理ユニット

(LUN) みたいなもので、名前空間でブロックプールを使うのはファイルシステムボリュームのようなものだ。

この手法には、スケーラビリティを確保する上で数々の利点がある。異なるアプリケーションで使う名前空間を分離でき、クラスタの全体的な可用性を向上できるのだ。ブロックプールによる抽象化のおかげで、ブロックストレージを使う他のサービスでは別の名前空間構造を使えるようになる。スケーリングのために他の手法も模索しており、たとえばメモリ内に保持するのを名前空間の一部だけにしたり NameNode を真に分散型の実装にしたりといった計画もある。

アプリケーションは、単一の名前空間を使いたがるので、名前空間をマウントして統一したビューを作ることもできる。それを効率的に行うには、クライアント側のマウントテーブルのほうがサーバー側のマウントテーブルより向いている。サーバー側に中央マウントテーブルを置く場合に比べてリモートプロシージャ呼び出しを回避できるし、耐障害性も高い。一番シンプルな手法は、クラスタ全体で名前空間を共有することだ。これを実現するには、クラスタ内の各クライアントに共通のクライアント側マウントテーブルを配布する。このマウントテーブルでは、各アプリケーションがプライベートな名前空間ビューを作れるようにもなっている。これは、分散システム内でのリモート実行をするためのプロセス単位の名前空間のようなものだ [PPT+93, Rad94, RP93]。

9.5 教訓

Hadoop ファイルシステムを開発したのはとても小さなチームで、彼らがシステムを安定させ、実運用に耐える頑健性を実現した。その成功の大半は、非常にシンプルなアーキテクチャに起因する。ブロックのレプリケート、定期的なブロックレポート、そして中央管理型のメタデータサーバーなどといったものだ。POSIX の仕様を完全に実現しようとはしなかったことも助けとなった。メタデータ全体をメモリ内に保持するために名前空間のスケーラビリティに制約が生じるが、そのおかげで NameNode がとてもシンプルになった。ふつうのファイルシステムにありがちな、複雑なロックの問題を回避できたのだ。Hadoop の成功にはそれ以外にも理由がある。早いうちから Yahoo!の本番環境に投入されたこともそのひとつで、素早くインクリメンタルな改良に貢献した。このファイルシステムは非常に頑健で、NameNode の障害はめったに発生しない。実際、ダウンタイムの大半はソフトウェアのアップグレードによるものである。フェイルオーバーのソリューション (手動だけどね) が登場したのも、つい最近のことだった。

スケーラブルなファイルシステムを Java で作るという選択に驚いた人も多かっただろう。確かに Java で NameNode をスケールさせるのは難しかった。オブジェクトメモリのオーバーヘッドやガベージコレクションがその原因だ。しかし、Java を選んだおかげで頑健性を確保できた。ポインタやメモリ管理のバグに悩まされることがなくなったのだ。

9.6 謝辞

Yahoo!による Hadoop への投資に感謝する。また、Hadoop をオープンソースで公開し続けてくれていることにも感謝する。HDFS や MapReduce のコードの 80%は Yahoo!が開発したものだ。すべての Hadoop コミッターやその他の協力者からの貢献にも感謝する。

第10章

Jitsi

Emil Ivov

Jitsi はビデオ通話、音声通話、デスクトップ共有、ファイル交換、メッセージングなどを行うアプリケーションである。さらに重要な事は多くのプロトコルを使ってこれらを実装していることである。実装しているプロトコルには、XMPP(Extensible Messaging and Presence Protocol) や SIP(Session Initiation Protocol) のような標準プロトコルから Yahoo!や Windows Live Messenger(MSN) のような専用プロトコルまで含まれる。また、Microsoft Windows や Apple Mac OS X、Linux、FreeBSD 上で動作する。多くの部分を Java で記述しているが、ネイティブで記述している部分もある。本章では、Jitsi の OSGi ベース・アーキテクチャに注目する。プロトコルをどのように実装および管理しているのか、構築する上で学んだことも振り返る [1]。

10.1 Jitsi の設計

マルチプロトコル・サポート、クロスプラットフォーム、開発者フレンドリーの 3 点が、Jitsi(かつて SIP Communicator と呼ばれた) の設計に際して制約として重視している項目である。

開発者の視点からみると、マルチプロトコルを採用するということは、すべてのプロトコルに対して共通のインタフェースが必要になる事を意味する。具体的には、ユーザがメッセージを送信するときには、グラフィカル・ユーザ・インタフェースは、常に `sendMessage` というメソッドを呼び出す必要がある。実際には、使用しているプロトコルによって、`sendXmppMessage` が呼ばれたり、`sendSipMsg` が呼ばれたりする。

我々のソースコードの多くが Java で記述されているという事実は、かなりの部分、2 点目の制約：クロスプラットフォーム　を満たしている。しかし、Java Runtime Environment (JRE) がサポートしていなかったり、我々が望むような方法で実装されていない部分もある。例え

[1] ソースコードの参照は、`http://jitsi.org/source`。Eclipse や NetBeans を使っているのなら、次のサイトにコンフィグレーション方法が載っている：`http://jitsi.org/eclipse` 、`http://jitsi.org/netbeans`

ば、ウェブカムからのビデオキャプチャのようなもの SIP Communicator である。そのために、Windows では DirectShow を使ったり Mac OS X では QTKit を使ったり、Linux では Video for Linux 2 を使ったりしている。プロトコルの場合と同じようにビデオ電話を制御する部分のソースコードは、これらの詳細部を隠蔽する (かなり複雑ではある)。

最後の開発者フレンドリーとは、新しい機能の追加が容易であるべきという事を意味する。今日、大勢の人々が VoIP を使っているが、使い方は様々である。多くのサービス・プロバイダやサーバ・ベンダは、異なるユースケースやアイデアを用いて新しい機能を追加している。Jisti を使う人にとって、これらの望まれる機能の実装が簡単で無ければばならない。何か新しい機能を追加する人が、追加・変更に関わる部分のソースコードだけを読んで理解できる必要がある。同様に、ある人の変更が他の人の作業に与える影響が最小限でなければばならない。

要約すれば、ソースコードの各部分は、それぞれ独立しているという環境が必要である。オペレーティング・システムに依存している部分が容易に置き換え可能でなければばならない；プロトコルのように同時に複数が動作しても同じように動作しなければばならない。また、各部分が完全にリライト可能でかつ、残りの部分は変更不要でなければばならない。最後に、インターネット経由でダウンロードしてプラグインとして追加できる機能や各部分を簡単にオン・オフできる機能も望まれる。

我々は、単純に我々自身でフレームワークを記述することを考えた、しかし、直ぐにそのアイデアを捨てた。我々は、できるだけ早く VoIP と IM のソースコードの記述を開始することを望んだ。プラグイン・フレームワークに数ヶ月間費やしたが、エキサイティングには思えなかった。誰かが OSGi の採用を提案したとき、これが完全にフィットするように思えた。

10.2　Jitsi と OSGi フレームワーク

OSGi の解説本は既にある。よって、このフレームワークの全体を説明するつもりは無い。代わりに、このフレームワークから得られる事および Jitsi での使われ方を説明する。

OSGi において最も大切な事はモジュールである。OSGi アプリケーションの機能はバンドルに分割されている。ひとつの OSGi バンドルは、Java ライブラリや Java アプリケーションを配布するときに使われる標準の JAR ファイルよりも小さい単位である。Jitsi は、これらのバンドルの集合である。 Windows Live Messenger と接続する責任を持つバンドルもあれば、XMPP と接続する責任を持つバンドルもある。ほかにも、GUI を扱うバンドルなどもある。これらすべてのバンドルは与えられた環境で動作する。我々の場合は、Apache Felix というオープンソースの OSGi 実装の環境下で動作する。

これらのモジュールは全て一緒に動作する必要がある。GUI バンドルは、プロトコル・バンドルを通じてメッセージを送信する必要がある。さらにメッセージ履歴を取り扱うバンドルを経由してこれらのメッセージを保存する必要がある。これは、OSGi サービスが何であるかを示している：OSGi サービスはバンドルの一部を表していて他バンドルからアクセス可能である。OSGi サービスは、ログ、ネットワーク経由のメッセージ送信、通話履歴の読出

しのような特定機能の利用を許可する Java インタフェースのグループである。実際に機能を実装するクラスは、サービス実装として知られている。サービス実装の多くは実装したサービス・インタフェース名を携える。サービス実装名は、“Impl” の接尾辞を持っている (例えば, ConfigurationServiceImpl)。OSGi フレームワークは開発者にサービスの実装を隠蔽している。また、OSGi フレームワークは、サービス実装がバンドル自身の外側には決して見えないことを確実にしている。この様にして、他のバンドルは、サービス・インタフェースを通してのみ利用できる。

ほとんどのバンドルはアクティベータも持つ。アクティベータは、start と stop メソッドを定義したシンプルなインタフェースである。Felix が Jitsi のバンドルをロードやリムーブする度に、バンドルが起動やシャットダウンの準備ができるように、これらのメソッドをコールする。これらのメソッドをコールするとき、Felix は BundleContext と言う名前のパラメータを渡す。BundleContext は、バンドルに OSGi 環境に接続する方法を与える。この様にして、使用したい OSGi サービスが何であろうと見つける事ができたり、自身を登録したりできる (図 10.1)。

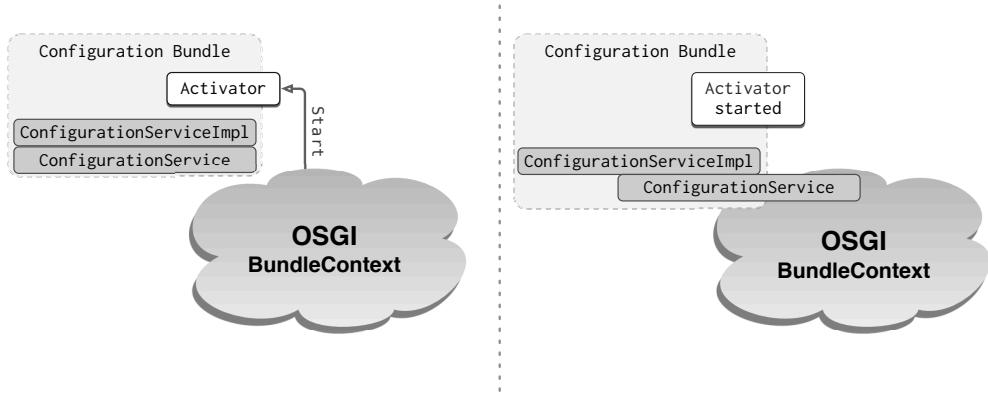



図 10.1: OSGi バンドル・アクティベーション

それでは、実際にこれがどのように動作するのかを見ていこう。プロパティを保存したり引き出したりするだけのサービスを想像してみよう。Jitsi では、これを ConfigurationService と呼んでいて次のようなものだ：

```
package net.java.sip.communicator.service.configuration;

public interface ConfigurationService
{
  public void setProperty(String propertyName, Object property);
  public Object getProperty(String propertyName);
}
```

極めてシンプルな ConfigurationService の実装は次のようなものである：

```
package net.java.sip.communicator.impl.configuration;

import java.util.*;
import net.java.sip.communicator.service.configuration.*;

public class ConfigurationServiceImpl implements ConfigurationService
{
  private final Properties properties = new Properties();

  public Object getProperty(String name)
  {
    return properties.get(name);
  }

  public void setProperty(String name, Object value)
  {
    properties.setProperty(name, value.toString());
  }
}
```

net.java.sip.communicator.service パッケージの中でサービスがどのように定義されるか注目すること。また、実装は net.java.sip.communicator.impl にある。Jitsi におけるすべてのサービスと実装は、このように２つのパッケージに分割される。OSGi は、バンドル自身が含まれる JAR の外部に対して、いくつかのパッケージだけを可視化する事を許可する。こうして、この分割は、バンドルに対してサービス・パッケージだけを *export* して実装を隠蔽する事ができる。

ユーザが、我々の実装を使い始めるのに必要となる最後の作業は、BundleContext に登録する事と ConfigurationService の実装を備えている事を、伝える事である。次は、これがどのように行われるかを示している：

```
package net.java.sip.communicator.impl.configuration;

import org.osgi.framework.*;
import net.java.sip.communicator.service.configuration;

public class ConfigActivator implements BundleActivator
{
  public void start(BundleContext bc) throws Exception
  {
```

```
    bc.registerService(ConfigurationService.class.getName(), // service name
         new ConfigurationServiceImpl(), // service implementation
         null);
  }
}
```

ConfigurationServiceImpl クラスが BundleContext に登録されると他のバンドルが使い始める事ができる。いくつかのランダム・バンドルがコンフィグレーション・サービスを使う例を示す：

```
package net.java.sip.communicator.plugin.randombundle;

import org.osgi.framework.*;
import net.java.sip.communicator.service.configuration.*;

public class RandomBundleActivator implements BundleActivator
{
  public void start(BundleContext bc) throws Exception
  {
    ServiceReference cRef = bc.getServiceReference(
                          ConfigurationService.class.getName());
    configService = (ConfigurationService) bc.getService(cRef);

    // And that's all! We have a reference to the service implementation
    // and we are ready to start saving properties:
    configService.setProperty("propertyName", "propertyValue");
  }
}
```

再びパッケージに注目する。net.java.sip.communicator.plugin の中で他によって定義されたサービスを使うバンドルをキープするが、自身をエクスポートしたりインプリメントしたりはしない。コンフィグレーション・フォームは、このようなプラグインの良い例である：これらは、ユーザにアプリケーションのある部分をコンフィグする事を許可する Jitsi ユーザインタフェースへの追加である。ユーザが優先権を変更するとコンフィグレーション・フォームは、ConfigurationService や、この機能に対する責任を持つバンドルと直接に作用し合う。しかし、他のバンドルは、作用する必要は無い (図 10.2)。

10.3　バンドルの構築と実行

バンドルのコード記述方法を一通り見てきた所で、次はパッケージングについて説明しよう。実行中、すべてのバンドルは OSGi 環境に次の 3 つを示す：他が利用可能な Java パッケージ (すなわちエクスポート・パッケージ)、他が使いたいと思う Java パッケージ (すなわちインポート・パッケージ)、BundleActivator クラスの名前。バンドルは、これを自身が配置される JAR ファイルのマニフェストを通して行う。

上記で定義した ConfigurationService に対するマニフェスト・ファイルは次のようになる：

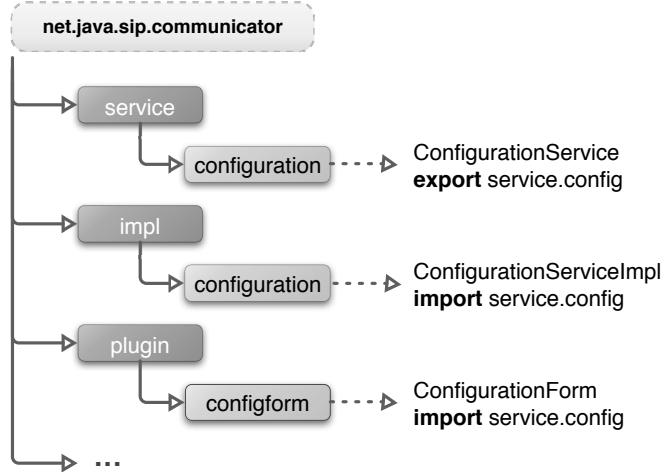



図 10.2: サービス・ストラクチャ

```
Bundle-Activator: net.java.sip.communicator.impl.configuration.ConfigActivator
Bundle-Name: Configuration Service Implementation
Bundle-Description: A bundle that offers configuration utilities
Bundle-Vendor: jitsi.org
Bundle-Version: 0.0.1
System-Bundle: yes
Import-Package: org.osgi.framework,
Export-Package: net.java.sip.communicator.service.configuration
```

JAR マニフェストが生成されると、バンドルを生成する準備ができる。Jitsi では、構築関連のタスクには Apache Ant を使う。バンドルを Jitsi ビルド・プロセスに追加するために、プロジェクトのルート・ディレクトリにある build.xml を編集する必要がある。バンドル JAR は、build.xml ファイルの最後の bundle-xxx ターゲットによって生成される。コンフィグレーション・サービスを構築するためには、次のようにする：

```
<target name="bundle-configuration">
  <jar destfile="${bundles.dest}/configuration.jar" manifest=
    "${src}/net/java/sip/communicator/impl/configuration/conf.manifest.mf" >

    <zipfileset dir="${dest}/net/java/sip/communicator/service/configuration"
        prefix="net/java/sip/communicator/service/configuration"/>
    <zipfileset dir="${dest}/net/java/sip/communicator/impl/configuration"
        prefix="net/java/sip/communicator/impl/configuration" />
  </jar>
</target>
```

お分かりのように、Ant ターゲットは、単純にコンフィグレーション・マニフェストを使って JAR ファイルを生成して、service と impl の階層構造から構成されるコンフィグレーション・パッケージに追加する。我々が必要な事は Felix にロードさせる事だけである。

Jitsi は単に OSGi バンドルの集合であることは既に説明した。ユーザがアプリケーションを実行する時、OSGi バンドルは、ロードする必要のあるバンドルのリストとともに、Felix をスタートさせる。 このリストは、`lib` ディレクトリの `felix.client.run.properties` ファイルの中にある。Felix は、スタートレベルによって定義される順番にバンドルをスタートする：すべての、あるレベルのバンドルは、続くレベルのバンドルがロードを開始する前に、ロードが完了していることを保証されている。このことは上記の例では確認できないが、我々のコンフィグレーション・サービスはプロパティをファイルに保存する。よって、*FileAccessService* を使う必要がある。これは、`fileaccess.jar` ファイルに収められている。ConfigurationService は、FileAccessService のあとに開始することを確実にする：

```
...
felix.auto.start.30= \
  reference:file:sc-bundles/fileaccess.jar

felix.auto.start.40= \
  reference:file:sc-bundles/configuration.jar \
  reference:file:sc-bundles/jmdnslib.jar \
  reference:file:sc-bundles/provdisc.jar \
...
```

`felix.client.run.properties` ファイルを見ると、先頭にパッケージのリストを見つける事ができる：

```
org.osgi.framework.system.packages.extra= \
  apple.awt; \
  com.apple.cocoa.application; \
  com.apple.cocoa.foundation; \
  com.apple.eawt; \
...
```

このリストは、Felix にシステムのクラス・パスからバンドルを利用するのに必要なパッケージは何であるかを教える。これは、このリスト上のパッケージは、他のバンドルによってエクスポートされる事なしに、バンドルによってインポートされうる (つまり、*Import-Package* マニフェスト・ヘッダに加える) 事を意味する。リストは、大抵、OS 依存の JRE 部品に由来するパッケージを含む。そして、Jitsi 開発者は、新しいパッケージを追加する必要はほとんどない；多くの場合、パッケージはバンドルによって利用される事ができる。

10.4　プロトコル・プロバイダ・サービス

Jitsi の `ProtocolProviderService` は、すべてのプロトコルの実装の振る舞いを定義する。これは、Jitsi が接続しているネットワーク上で、(ユーザ・インタフェースのような) 他のバンドルがメッセージの送受信、通話、ファイルシェアを行う必要があるときに使うインタフェースである。

プロトコル・サービス・インタフェースは、net.java.sip.communicator.service.protocol パッケージ下に見つける事ができる。サービスに対して複数の実装があったり、サポート・プロトコル毎のサービス実装があったりする。全ては、net.java.sip.communicator.impl.protocol.-protocol_name に保存されている。

service.protocol ディレクトリから始めよう。もっとも重要な部分は、*ProtocolProviderService* インタフェースである。プロトコル関連のタスクを行おうとすると、必ず BundleContext にあるサービスの実装を調べなくてはならない。サービスとその実装は、Jitsi にサポートするネットワーク、接続状態の確認や詳細確認、最も重要なチャットや通話のような実際の通信タスクを実装したクラスへの参照を取得する事を可能にする。

オペレーション・セット

前に触れたように、ProtocolProviderService は、様々な通信プロトコルとその差異を隠して利用するのに必要である。これは、メッセージ送信のようなすべてのプロトコルが持っている機能に対しては極めてシンプルであるが、サポートするプロトコルが少ない機能に対しては手の込んだものになる。これらの違いは、しばしばサービス自身に由来する：例えば、世の中にあるほとんどの SIP サービスは、連絡先リストをサポートしていないが、他のプロトコルでは大抵サポートしている。MSN と AIM は、もうひとつの分かりやすい例である：どちらのプロトコルもオフラインのユーザに対してのメッセージ送信はサポートしていないが、他のプロトコルではサポートしている (現在は変わった)。

ProtocolProviderService は、GUI 等の他のバンドルで行っているように、相違を扱う方法が必要である：実際に通話する機能がなければ、AIM コンタクトに通話ボタンを追加する意味がない。

OperationSets チェックシート (図 10.3)。当然の事だが、これらは操作のセットであり、Jitsi バンドルが、プロトコル実装を制御するのに使用するインタフェースを提供する。オペレーション・セット・インタフェースにあるメソッドは、すべて特定の機能に関連している。例えば、*OperationSetBasicInstantMessaging* は、インスタント・メッセージの送受信用のメソッドを持っていて、Jitsi が受信したメッセージを引き出すためのリスナーを登録する。もうひとつの例は、*OperationSetPresence* である。*OperationSetPresence* は、連絡先リスト上の状態を問い合わせたり、自分自身の状態を登録するメソッドを持っている。よって、GUI が状態を更新して、連絡先を表示したり、連絡先にメッセージを送信するとき、プレゼンスやメッセージングをサポートしようがしまいが、対応するプロバイダに最初に問い合わせる事ができる。ProtocolProviderService がこの目的で定義するメソッドは、次のようになる：

```
public Map<String, OperationSet> getSupportedOperationSets();
public <T extends OperationSet> T getOperationSet(Class<T> opsetClass);
```

OperationSets は、新しく追加したプロトコルが OperationSet で定義したオペレーションのいくつかしかサポートしないような事がないように設計されなければならない。例えば、相

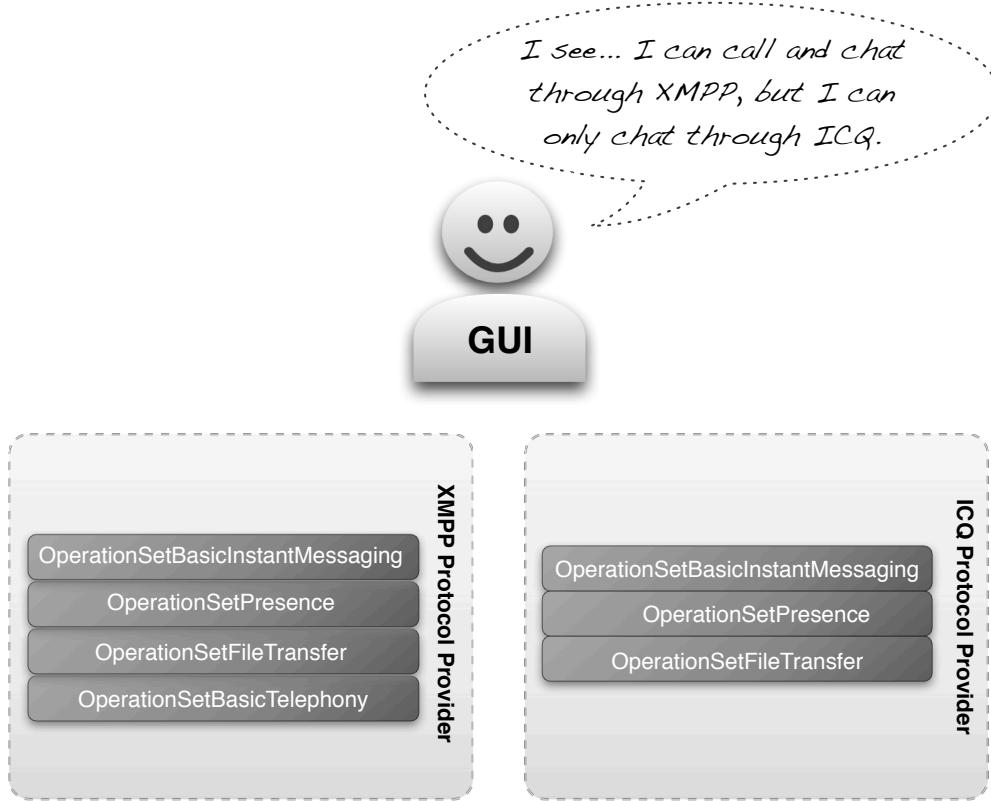



図 10.3: オペレーション・セット

互の状態を問い合わせる機能はあるが、サーバに連絡先を保存する機能をサポートしないプロトコルがある。よって、`OperationSetPresence` のプレゼンス管理と仲間リストの引き出し機能を結合するよりも、連絡先をオンラインで保存できるプロトコルとだけ使用できる `OperationSetPersistentPresence` を定義した方が良い。他方、受信することはせずに送信だけできるプロトコルもあり、送信メッセージと受信メッセージが安全に結合できる理由である。

アカウント、ファクトリおよびプロバイダのインスタンス

`ProtocolProviderService` の重要な特性は、インスタンスとプロトコル・アカウントは 1 対 1 に対応しているという事である。よって、常にユーザが登録したアカウントの数と同じ数の `BundleContext` のサービス実装がある。

ここで、誰がプロトコル・プロバイダを生成・登録するのだろうかと思うだろう。これには、2 つの異なるエンティティが関わっている。一つ目は、`ProtocolProviderFactory` である。これ

は、他のバンドルがプロバイダをインスタンス化して、サービスとして登録する事を許可する。プロトコル毎にファクトリがあり、各ファクトリは対応するプロトコルのプロバイダを生成する責任を持つ。ファクトリ実装は、プロトコル内部の残りの部分に保存される。SIP を例にすると、`net.java.sip.communicator.impl.protocol.sip.ProtocolProviderFactorySipImpl` である。

アカウント生成を含むふたつ目のエンティティは、プロトコル・ウィザードである。ファクトリとは違って、ウィザードグラフィカル・ユーザ・インタフェースを含むためプロトコル実装の残りの部分から分離されている。例えば、SIP アカウントを生成を許可するウィザードは、`net.java.sip.communicator.plugin.sipaccregwizz` にある。

10.5　メディア・サービス

IP 上でリアルタイム通信を行う場合、理解しておくべき重要な事がある：SIP や XMPP のようなプロトコルは、もっとも一般的な VoIP プロトコルとして認識されている一方、インターネット上で音声やビデオを実際に動かすプロトコルではない。これはリアルタイム・プロトコル (RTP) によって扱われる。SIP と XMPP は、RTP パケットの送信先アドレスの決定、音声、ビデオの符号化方式 (つまりコーデック) の交渉などの RTP が必要とするすべての準備に対してだけ責任を持つ。ユーザの位置管理、プレゼンス管理、着信音など他の多くの事に対しても面倒を見る。これは、SIP や XMPP のようなプロトコルはシグナリング・プロトコルと呼ばれる理由である。

これは、Jitsi のコンテキストではどんな意味をもつのだろうか？まず最初に、*sip* や *jabber* の Jitsi パッケージには、音声やビデオのフローを操作するソースコードは見つからないと言う事である。この手のソースコードは、MediaService にある。MediaService とその実装は、`net.java.sip.communicator.service.neomedia` と `net.java.sip.communicator.impl.neomedia` にある。

なぜ “neomedia”?

neomedia パッケージ名の “neo” は、初期に使っていた類似のパッケージを置き換えである事を意味し、完全に置き換えを行った。これは、我々の経験則「最新を完全に保つ所までアプリケーションを設計するのに多くの時間を使う事には価値が無い」を生んだ。単純に、すべてを考慮にいれる方法は無いので、後に変更される運命にある。さらに、綿密な設計フェーズは複雑性を産むが、準備したようなシナリオは決して発生しないために、使われることはない。

MediaService 自身に加えて、特に重要なインタフェースが二つある：MediaDevice と MediaStream である。

キャプチャ、ストリーミング、再生

MediaDevice は、通話時に使うキャプチャ・デバイスと再生デバイスを表わす (図 10.4)。マイクロフォン、スピーカ、ヘッドセット、ウェブカムは全てこのような MediaDevice の例であるが、これだけではない。Jitsi のデスクトップ・ストリーミングとシェアリング・コールは、デスクトップからビデオ・キャプチャを行う。会議通話は、参加者の音声を合成するために AudioMixer デバイスを使う。すべてのケースで、MediaDevice は単一の MediaType を表わす。つまり、音声かビデオのどちらかには成れるが両方にはなれない。これは、例えば、マイクロフォンが統合されたウェブカムを持っている場合、Jitsi はふたつのデバイスと認識する：ひとつはビデオ・キャプチャだけを行い、もうひとつはサウンド・キャプチャだけを行う。

デバイスだけでは、電話やビデオ通話を行うには、不十分である。メディアを再生したりキャプチャしたりするのに加えて、ネットワーク上に送信できなければならない。これには、MediaStream が使われる。MediaStream インタフェースは、MediaDevice と通話相手をつなげる。通話中に交換する通話相手との受信および送信パケットを表わす。

デバイスと同様に、ひとつのストリームは、一つの MediaType に対してのみ責任を持つ。これは、音声/ビデオ通話の場合、Jitsi はふたつの分離したメディア・ストリームを生成して、それぞれを対応する音声、ビデオの MediaDevice と接続する。

コーデック

メディア・ストリーミングで、もう一つの重要な概念は、コーデックとして知られている MediaFormat である。デフォルトでは、多くのオペレーティング・システムは、オーディオを 48KHz PCM か類似の方式でキャプチャする。これは、我々がしばしば、“raw audio” として参照するもので、WAV ファイルとして取得するオーディオ方式である。WAV ファイルは、高品質であるが莫大なサイズを持つ。この PCM フォーマットでインターネット上にオーディオを転送を試みるのは非現実的である。

これは、コーデックの意味を示す：オーディオやビデオを様々な方法で表現し転送する。iLBC や 8KHz Speex や G.729 のようなオーディオ・コーデックは、狭帯域であるが、こもったように聞こえる。ワイドバンド Speex や G.722 は、高品質のオーディオを提供するが、より多くの帯域を使用する。高い品質を保ちつつ帯域も合理的であることを狙ったコーデックもある。ポピュラーなビデオコーデックである H.264 が好例である。ここでのトレードオフは、変換時の計算量である。Jitsi で H.264 ビデオ通話を使用すると、高い品質の画像と合理的な帯域幅を痛感するだろう。しかし、CPU 負荷は高い。

単純化すると、コーデック選択は妥協が全てであるという事である。帯域幅、品質、CPU 消費量または、これらの組み合せのどれかを犠牲にする。ほとんどの場合、VoIP 関連の人はコーデックについて、これ以上知る必要は無い。

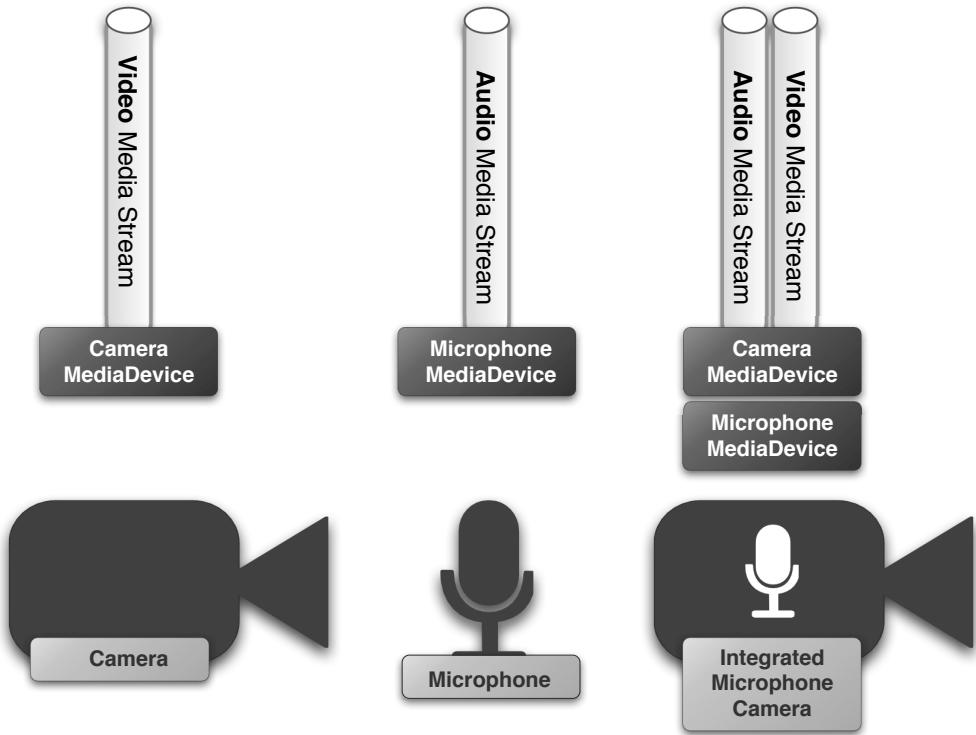



図 10.4: 異なるデバイスに対するメディア・ストリーム

プロトコル・プロバイダとの接続

オーディオ/ビデオをサポートしている Jitsi のプロトコルは全て、MediaService を全く同じ方法で使用する。最初にシステムで使用可能なデバイスを MediaService に問い合わせる：

```
public List<MediaDevice> getDevices(MediaType mediaType, MediaUseCase useCase);
```

MediaType はオーディオ・デバイスかビデオ・デバイスのどちらに興味があるのかを示す。MediaUseCase パラメータは、現在の所、ビデオ・デバイスの場合のみ扱われる。これは、次のようにしてメディア・サービスに利用できるデバイスを知らせる。通常通話 (MediaUseCase.CALL) の場合は、利用可能なウェブカムのリストを返し、デスクトップ・シェアリング・セッション (MediaUseCase.DESKTOP) の場合は、ユーザ・デスクトップへの参照を返す。

次のステップは、デバイスが利用できるフォーマットのリストを取得する事である。これは、`MediaDevice.getSupportedFormats` メソッドを使って次のようにする：

```
public List<MediaFormat> getSupportedFormats();
```

プロトコルの実装は、このリストを取得してリモート・パーティに送る。リモート・パーティは、このリストの中からサポートしているフォーマットのサブセットを作成して応答する。

この交換は、オファー/アンサーモデルとして知られていて、セッション記述プロトコルや同類のプロトコルで利用される。

フォーマット、ポート番号および IP アドレスを交換した後、VoIP プロトコルはメディア・ストリームの生成、コンフィグ、開始を行う。この初期化は、大雑把に次のような流れになる：

```
// 最初にストリーム・コネクタを生成する。ストリームコネクタはメディア・サービス
// に利用するソケットを教える。
// 利用するソケットは、メディア転送時の RTP、フロー制御や統計メッセージの RTCP である。
StreamConnector connector =  new DefaultStreamConnector(rtpSocket, rtcpSocket);
MediaStream stream = mediaService.createMediaStream(connector, device, control);

// MediaStreamTarget は、通信相手のメディアが使おうとするアドレスとポート番号を示す。
// この情報を交換する方法は、VoIP プロトコル毎に異なる。
stream.setTarget(target);

// MediaDirection パラメータは、stream に着信なのか発信なのか両方なのかを知らせる。
stream.setDirection(direction);

// そして、ストリーム・フォーマットを設定する。セッション交渉の応答に含まれるリストの
// 最初のフォーマットを使う。
stream.setFormat(format);

// 最後に、メディア・デバイスから media を取得する準備が整い、インターネット上に、
// ストリーミングする。
stream.start();
```

自分のウェブカムで wave でき、マイクを使って、「Hello world!」と言うことができる。

10.6 UI サービス

ここまでの所で、Jitsi におけるプロトコルの扱い、メッセージ送受信、通話の部分をカバーした。さらに、Jitsi は実際の一般の人々に使われるアプリケーションであり、ユーザインタフェースが最も重要な側面を持つ。多くの時間、ユーザインタフェースは Jitsi の他の全てのバンドルが明示するサービスを使う。しかし、そうとも限らない場面もある。

プラグインは、気に留めておくべき最初の例である。Jitsi におけるプラグインは、ユーザと相互作用できる必要がある。これは、プラグインは、ユーザ・インタフェースのウィンドウやパネルに、コンポーネントをオープンしたり、クローズしたり、ムーブしたり、追加する必要がある事を意味する。これは、UIService が実行される時の話である。Jitsi メイン・ウィンドウの基本制御を可能にすると供に、Mac OS X ドックのアイコンや Windows の通知エリアでアプリケーションをコントロールする方法である。

プラグインは、単純に連絡先リストを使うのに加えて、この機能を拡張できる。Jitsi の暗号チャット (OTR) をサポートするプラグインは、プラグインによる機能拡張の良い例である。OTR バンドルは、ユーザ・インタフェースの様々な部品の中からいくつかの GUI コンポー

ネントを登録する必要がある。チャット・ウィンドウに錠前ボタンを追加し、すべての連絡先に対する右クリックメニューにサブセクションを追加する。

良いニュースは、いくつかのメソッド呼び出しだけで、これを行う事ができるという事である。OTR バンドルに対する OSGi アクティベータ OtrActivator は、次のソースコードを含んでいる：

```
Hashtable<String, String> filter = new Hashtable<String, String>();

// Register the right-click menu item.
filter(Container.CONTAINER_ID,
    Container.CONTAINER_CONTACT_RIGHT_BUTTON_MENU.getID());

bundleContext.registerService(PluginComponent.class.getName(),
    new OtrMetaContactMenu(Container.CONTAINER_CONTACT_RIGHT_BUTTON_MENU),
    filter);

// Register the chat window menu bar item.
filter.put(Container.CONTAINER_ID,
           Container.CONTAINER_CHAT_MENU_BAR.getID());

bundleContext.registerService(PluginComponent.class.getName(),
           new OtrMetaContactMenu(Container.CONTAINER_CHAT_MENU_BAR),
           filter);
```

みてわかるように、グラフィカル・ユーザ・インタフェースへのコンポーネント追加は、単純に OSGi サービスの登録に行き着く。反面、UIService の実装は、PluginComponent インタフェースの実装を探索する。新しい実装が登録された事を検出すると直ぐに、実装への参照を取得し、OSGi サービス・フィルタに示されたコンテナに追加する。

右クリック・メニュー項目のときに、これがどのように起こるのかを示す。UI バンドル内で、右クリック・メニューを表わすクラス MetaContactRightButtonMenu は、次のソースコードを含んでいる：

```
// Search for plugin components registered through the OSGI bundle context.
ServiceReference[] serRefs = null;

String osgiFilter = "("
    + Container.CONTAINER_ID
    + "="+Container.CONTAINER_CONTACT_RIGHT_BUTTON_MENU.getID()+")";

serRefs = GuiActivator.bundleContext.getServiceReferences(
        PluginComponent.class.getName(),
        osgiFilter);
// Go through all the plugins we found and add them to the menu.
for (int i = 0; i < serRefs.length; i ++)
{
    PluginComponent component = (PluginComponent) GuiActivator
        .bundleContext.getService(serRefs[i]);
```

```
    component.setCurrentContact(metaContact);

    if (component.getComponent() == null)
        continue;

    this.add((Component)component.getComponent());
}
```

これで全てである。Jitsi 内にあるウィンドウの多くは、全く同じ事をする：PluginComponet インタフェースを実装するサービスに対するバンドル・コンテキストを探索する。このインタフェースはフィルタを持っていて、このフィルタは対応するコンテナに追加されたい事を示す。プラグインは、行き先を示すボードを持っているヒッチハイカーのようなものである。Jitsi ウィンドウが彼らを拾う運転手である。

10.7 得た教訓

SIP Communicator に取り掛かったときに、最も多かった批判、疑問は、次のようなものである：“なぜ Java を使っている?、動作が遅いでしょう？、音声通話やビデオ通話での音声品質は、たかが知れている！”。“Java は遅い” という通説は、Jitsi を試さずに Skype を使い続けるとめの理由として、潜在的なユーザによって、繰り返し述べられている。しかし、このプロジェクトにおける我々が最初に学んだ教訓は、Java における効率性への関心は、既に C++ や他のネイティブ言語における関心と同程度のものである、という事である。

全ての項目を厳密に分析した結果、Java を選択したというつもりはない。我々は、シンプルにに Windows や Linux 上で実行する簡単な方法を望んだ。そして、Java と Java Media Framework は、これを行う比較的簡単な方法のように思えた。

この決定を悔やむ理由は、今の所、ほとんどない。それどころか、完全に透過的というわけではないが、Java は、移植性が高く、SIP Communicator のソースコードの 90%は、OS 間で共通である。これは、かなり複雑にも係わらず、すべてのプロトコル・スタック (たとえば、SIP、XMPP、RTP など) の実装を含んでいる。OS に依存するようなソースコード部分に心配を払う必要は無く、大変便利という事が証明された。

さらに、Java の高い評判は、コミュニティを形成する上で大変重要である事が分かった。コントリビュータは希少資源である。人々は、アプリケーションの性質を気に入る必要があり、時間とモチベーションを見つける必要がある—これらのすべてを集めるのは困難である。よって、新しい言語を習う必要がないというのは、利点である。

多くの期待に反して、Java の実効速度が遅いという仮説は、ネイティブ言語に移行する理由にはなっていない。多くの場合、ネイティブ言語にする決断は、OS との融合と Java がどれだけ OS 依存のユーティリティにアクセスできるかに掛かっている。以下で、Java が不十分である 3 大領域について説明する。

Java サウンド対 PortAudio

Java サウンドは、オーディオを取り込んだり、再生したりする Java のデフォルト API である。これは、Java ランタイム環境の一部であるので、Java 仮想マシン上のすべてのプラットフォームで動作する。SIP Communicator としての最初の数年、Jitsi は、もっぱら JavaSound を使っていて、かなり不便であった。

まず第一に、この API は使用するオーディオ・デバイスを選択する方法を提供しなかった。これは大問題である。コンピュータを音声やビデオ通話に使うとき、ユーザは、よく先進の USB ヘッドセットや、他の高品質なオーディオ・デバイスを使う。コンピュータ上で、複数のデバイスが存在するとき、JavaSound は OS がデフォルトと考えるデバイスを辿るが、多くの場合良くない。デフォルトのサウンドカード上のスピーカーを通して音楽を聞きながら、という風に多くのユーザは、他のアプリケーションを走らせながら使う事を好む。さらに重要なことは、多くの場合、SIP Communicator にとって、オーディオ通知を一つのデバイスに送り、実際の通話音声は他のデバイスに送るのが都合が良い。つまり、着信通知をコンピュータの前にないとしてもユーザに聞こえるようにスピーカーから流し、着信に応答したらヘッドセット側に切り替える。

これは、Java Sound では不可能である。さらに Linux の実装では、今日の Linux ディストリビューションでは廃止予定の OSS を使っている。

我々は、他のオーディオシステムを使うことを決定した。我々は、マルチプラットフォームを諦めたくなかったので、できれば我々自身での実装は避けたかった。これは、PortAudio[2] が特に得意とする事である。

Java で実現できないことは、クロスプラットフォームのオープンソースプロジェクトを使うのが次善の策である。PortAudio への切り替えは、上記のようなオーディオのレンダリングや取り込みに関する優れたサポートを提供する。また、Windows、Linux、Mac OS X で動作し、FreeBSD など我々がパッケージを提供できないでいる OS もサポートしている。

ビデオ・キャプチャ、レンダリング

ビデオは、オーディオと同様に重要である。しかし、これは、Java クリエータにとっては同様では無いようである。なぜなら、ビデオをキャプチャしたり、レンダリングしたりするデフォルトの JRE API が無いからである。Sun がメンテナンスを止めるまで、しばらく、Java Media Framework がこのような API を目指していたようである。

自然に、我々は PortAudio 方式のビデオの代替手段を探し始めた。しかし、今回はダメだった。最初は、Ken Larson[3] の LTI-CIVIL フレームワークに決め掛けた。これは、すばらしいプロジェクトで、しばらく[4] 使用した。しかし、リアルタイム通信のコンテキストで使うには最適というわけではない事がわかった。

---

[2] http://portaudio.com/
[3] http://lti-civil.org/
[4] 実際に、まだ、非デフォルトのオプションとして残っている

Jitsi でのビデオ通信を満足に行う唯一の方法は、我々自身でネイティブの取り込みやレンダリングを実装する事だった。これは、簡単な決断ではなかった。複雑性が大幅に増し、潜在的なメンテナンス負荷がプロジェクトに追加される。しかし、他に選択肢はなかった。我々は高品質のビデオ通話を望んだ。そして実現した！

我々のネイティブの取り込みとレンダリングは、Linux、Mac OS X、Windows でそれぞれ、Video4Linux 2、QTKit、DirectShow/Direct3D を直に使った。

ビデオのエンコーディングとデコーディング

SIP Communicator 従って Jitsi は、最初のリリースからビデオ通話をサポートした。これは、Java Media Framework は、H.263 コーデックと 176x144(CIF) フォーマットをサポートするからである。H.264 CIF の見た目を知っている人は笑うだろう。現在、もしこれしか提供できないのであれば、ビデオ・チャット アプリケーションを使う人はいないだろう。

より高い品質を提供するためには、FFmpeg のような他のライブラリを使う必要があった。ビデオ・エンコーディングは、Java が性能的な限界を示す数少ない分野である。よって、FFmpeg 開発者がビデオを最も効率的な方法で処理するために実際にアセンブラを多くの場所で使っているという事実が示すように、我々も他の言語を使う。

その他

より良い結果を得るために、ネイティブにする事を決定した箇所が多くある。Mac OS X における Systray 通知の Growl、Linux における libnotify は、こういった例である。他には、Microsoft Outlook や Apple Address Book の連絡先データベースへの問い合わせ、相手先アドレスによる送信元 IP アドレスの決定、既にある Speex や G.722 のコーデック実装の利用、デスクトップ・スクリーンショットの取得、キャラクタのキーコードへの変換がある。

重要な事は、我々がネイティブに解決策を求めようと思えば可能であり、実際に行ったという事である。これは、次のポイントをもたらした。Jitsi を初めて以来、多くの部分を修正したり、追加したり、完全に書き換えたりしている。これは、見た目、使い心地、性能をより良いものにしたかったからである。しかし、最初によいものを出せなかった事を後悔はしていない。疑いを持ったら、選択肢の中から可能なものを選び、実行した。より良い方法を知るまで待つ事もできたが、これを行っていたら、今日までに Jitsi は存在していないだろう。

10.8　謝辞

本章の全ての図を作成してくれた Yana Stamcheva に感謝する。

第 11 章

LLVM

Chris Lattner

本章では、LLVM [1] を形づくるもとになった設計上の判断について議論する。LLVM は包括的なプロジェクトで、各種の低レベルツールコンポーネント (アセンブラやコンパイラ、デバッガなど) が密接に結びついている。また、主に Unix システム上で使われている既存のツール群との互換性を考慮している。“LLVM” という名前は元々は略語だったが、今ではこのプロジェクトのブランドになっている。LLVM は独特の機能を提供するし、すばらしいツール (たとえば Clang コンパイラ [2]。これは C/C++/Objective-C のコンパイラで、GCC に比べていろんな利点がある) が含まれていることでも有名だが、LLVM がその他のコンパイラと一線を画す最大の要因は、その内部アーキテクチャだ。

LLVM が生まれたのは 2000 年 12 月のこと。再利用可能なライブラリとよくできたインターフェイスをまとめたものとして設計された [LA04]。その頃のオープンソースなプログラミング言語の実装は、特化型のツールとして設計されていることが多く、たいていは一枚岩の大きな実行ファイルだった。たとえば、GCC などの静的なコンパイラの中からパーサーだけを再利用して静的解析やリファクタリングを施そうとしても、それはとても難しかったのだ。スクリプト言語の中にはそのランタイムを別のアプリケーションに組み込めるようになっているものも多かったが、そのランタイムだって一枚岩なひと固まりのコードで、それをインクルードするかしないかというだけのことだった。一部だけを再利用する手段はなくて、言語をまたがって実装を共有するプロジェクトなどほとんどなかった。

コンパイラ自体の構成以上に、主要な言語実装を取り巻くコミュニティもまた偏向していた。GCC や Free Pascal そして Free BASIC のような昔ながらの静的コンパイラか、またはインタプリタや Just-In-Time (JIT) コンパイラのような実行時コンパイラ。その**どちらかだけ**を提供するというところが多かった。両方に対応した言語実装はほとんどなかったし、仮にあったとしても、両方で共有しているコードはごく一部だけだった。

この 10 年で、LLVM がその状況を大きく変えた。LLVM は今や共通の基盤として使われるようになり、静的コンパイル型から実行時コンパイル型まで幅広い言語の実装に使われて

[1] `http://llvm.org`
[2] `http://clang.llvm.org`

いる (GCC がサポートする言語群の他にも、Java や.NET、Python、Ruby、Scheme、Haskell、D など。あまり知られていないものも含めると、さらに多くの言語に対応する)。また、さまざまな特化型コンパイラにとって代わるようになってきた。Apple の OpenGL スタック内の実行時特殊化エンジンや、Adobe After Effects の画像処理ライブラリなどだ。LLVM はさまざまな分野の新製品開発にも使われるようになっている。中でも一番有名なのは、GPU プログラミング言語 OpenCL とそのランタイムだろう。

11.1 古典的なコンパイラの設計

昔ながらの静的コンパイラ (たいていの C コンパイラ) でよくある設計は、フロントエンド・オプティマイザ・バックエンドの 3 フェーズにわけるものだ (図 11.1)。フロントエンドは、ソースコードを解析してエラーをチェックし、言語固有の抽象構文木 (AST) を作ってコードを表現する。AST は最適化に備えて別の表現に変換されることもある。オプティマイザやバックエンドは、この表現上で動く。

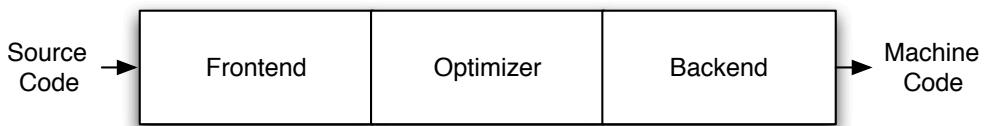



図 11.1: 3 フェーズコンパイラの主要なコンポーネント

オプティマイザの役割は、さまざまな変換 (冗長な計算の除去など) を行ってコードの実行時間を短縮することだ。通常は、言語やターゲットとは独立している。バックエンド (コードジェネレータとも呼ばれる) は、そのコードをターゲット上の命令セットにマップする。単に**正しい**コードを作るだけではなく、より**よい**コードを作ってそのアーキテクチャ独自の機能を活用できるようにするのもバックエンドの役割だ。コンパイラのバックエンドでの一般的な処理は、命令の選択やレジスタの確保、命令のスケジューリングなどだ。

このモデルは、インタプリタや JIT コンパイラにも同様にあてはまる。Java 仮想マシン (JVM) もこのモデルの実装で、フロントエンドとオプティマイザの間のインターフェイスとして Java バイトコードを使っている。

この設計の実装

この古典的な設計の重要な利点は、コンパイラで複数のソース言語やターゲットアーキテクチャをサポートするようになったときに効いてくる。オプティマイザ内で共通のコード表現を使うようにしてすれば、どんな言語のフロントエンドだってその形式に変換するように

書いておける。一方バックエンド側も、共通の表現から任意のターゲットへのコンパイルを書けるようになる。その様子を図 11.2 に示した。

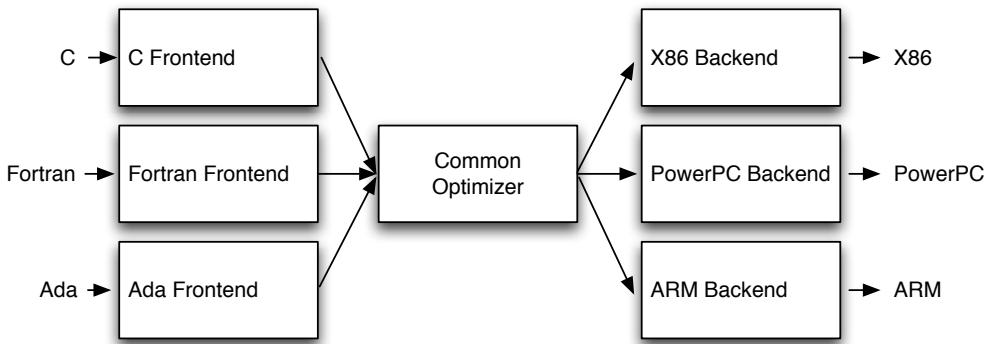



図 11.2: 再利用性

この設計のコンパイラで新たなソース言語 (Algol や BASIC など) に対応するために必要なのは、それ用のフロントエンドを実装することだ。オプティマイザやバックエンドは、既存のものを流用できる。もしこれらのフェーズを分割していなければ、新たな言語をサポートしようとするたびにスクラッチですべて書かないといけないことになってしまう。N 種類のターゲットと M 種類のソース言語をサポートするには、N*M 種類のコンパイラが必要になるということだ。

3 フェーズ設計には別の利点もある (再利用性から直接導けるものだ)。それは、単一のソース言語と単一のターゲットしかサポートしていないとしても、そのコンパイラがより幅広いプログラマーの役に立てるということだ。オープンソースプロジェクトにとっては、これはつまり、より多くの貢献者を巻き込めるようになるということだ。当然、そのほうがコンパイラの機能追加や改良も進めやすくなる。これこそが、多くのコミュニティで使われるオープンソースのコンパイラ (GCC など) のほうが FreePASCAL のような限定的なコンパイラよりも高性能のオプティマイザを作れる理由でもある。プロプライエタリなコンパイラの場合はその限りではない。コンパイラの品質は、純粋にそのプロジェクトの予算に絡んでくる。たとえば Intel ICC Compiler が生成するコードが高品質であることは広く知られているが、このコンパイラの利用者はそんなに多くない。

3 フェーズ設計の主要な利点として最後にもう一つあげられるのが、各フェーズの実装に求められるスキルがそれぞれ異なるという点だ。これらを分割しておくことで、「他はともかくフロントエンドならわかるよ」といおう人でもコンパイラに手を加えたり保守したりできるようになる。技術的な問題というよりはソーシャルな問題ではあるが、現実的にはこのあたりが大きく影響してくる。特にオープンソースプロジェクトの場合、参入障壁はできるだけ低くしておきたいものだ。

11.2　既存の言語の実装

3 フェーズコンパイラには利点があることは認めざるを得ないし、実際コンパイラの教科書にもそう書かれている。でも実際のところ、そのメリットを完全に理解している人は少ない。オープンソースの言語の実装を (LLVM 以前にさかのぼって) 見てみると、わかるはずだ。Perl や Python、Ruby、そして Java。これらの言語間では一切コードを共有していない。さらに、Glasgow Haskell Compiler (GHC) や FreeBASIC といったプロジェクトはさまざまな CPU に対応した再利用性を持っているが、その実装はたった一つのソース言語しかサポートしていない。特化型のコンパイラ技術はいろいろ公開されており、画像処理や正規表現、グラフィックカードのドライバ用などの CPU 固有の作業用に JIT コンパイラが実装されている。

とはいえ、このモデルを採用してコードの共有に成功した例もある。ここでは三つの有名な例を取り上げよう。まずは Java や.NET の仮想マシンだ。これらのシステムでは、JIT コンパイラやランタイムサポート、そしてうまく考えられたバイトコードフォーマットを提供する。つまり、どんな言語であっても、このバイトコードフォーマットへの変換さえできれば (実際、できている言語がいろいろある [3])、オプティマイザだけでなく JIT やランタイムも活用できるということだ。その代償として、ランタイムの選択によって実装の柔軟性が縛られるという点がある。JIT コンパイルやガベージコレクション、そして採用するオブジェクトモデルなどの制約を受けるということだ。その結果、このオブジェクトモデルにマッチしない言語、たとえば C などを (LLVM プロジェクトで) コンパイルするときには、パフォーマンスが二の次になってしまう。

二番目の例は、おそらく最も不運なものだ。しかし、コンパイラの技術の再利用という点では最も有名な例でもある。入力のソースを C のコード (あるいは別の言語のコード) に変換して、既存の C コンパイラに渡すというものだ。これによって、オプティマイザやコードジェネレータの再利用ができるようになり、柔軟性も上がり、ランタイムも制御できるようになる。実装する側にとっては、フロントエンドの実装がいちばんわかりやすいし、実装や保守も楽だ。残念ながら、この方式だと例外処理の実装が非効率的になるし、デバッグ機能も貧弱になる。コンパイル速度も遅くなるし、末尾再帰の保証 (などの C でサポートしていない機能) を要求する言語で問題になる。

このモデルの実装での成功例として最後に取り上げるのが、GCC[4] だ。GCC はさまざまなフロントエンドとバックエンドに対応しており、幅広いコミュニティが活発に活動している。GCC は、C コンパイラとしての長い歴史があり、複数のターゲットに対応している。そこにちょっとした手を加えて、いくつかの他の言語も組み込まれていた。時を経て、GCC コミュニティは徐々に設計を改良しはじめた。GCC 4.4 の時点ではオプティマイザの内部表現が新しい形式 (“GIMPLE Tuples”) になって、以前よりもフロントエンドの表現と切り離されたものになった。また、Fortran や Ada 用のフロントエンドでは、クリーンな AST を使っている。

これら三つの例は大成功をしたものだが、どのアプローチにも使い方に大きな制限がある。

[3] http://en.wikipedia.org/wiki/List_of_JVM_languages
[4] 後付けで、今は “GNU Compiler Collection” の略だということになっている

もともとモノリシックなアプリケーション用に作られているからだ。たとえば、GCC を他のアプリケーションに組み込んだりするのは現実的に不可能だろう。同じく、GCC を JIT コンパイラとして使ったり GCC の一部分だけを取り出して再利用したりといったことも不可能だ。GCC の C++フロントエンドだけを使ってドキュメントの生成やコードのインデキシング、リファクタリング、静的解析などがしたいという人がいたとしても、GCC 全体をモノリシックなアプリケーションとして使うしかない。あるいはプラグインを書いて、外部のコードを GCC のプロセスに注入するということになるだろう。

GCC の一部だけを切り出してライブラリとして再利用できないのには、複数の理由がある。グローバル変数を使いまくっていること、不変条件の強要が弱いこと、データ構造の設計がまずいこと、コードが乱雑なこと。さらに、マクロを使っているせいで、複数のフロントエンドとバックエンドのペアを同時にサポートするようにコンパイルすることができない。中でも一番修正しにくいのは、初期の設計や当時の時代背景に由来する先天的なアーキテクチャの問題だ。特に、GCC はレイヤー化の問題と抽象化の漏れに苦しんでいる。バックエンドはフロントエンドの AST を使ってデバッグ情報を生成するし、フロントエンドはバックエンドのデータ構造を生成する。そしてコンパイラ全体がグローバルなデータ構造に依存しており、これはコマンドラインインターフェイスで準備するものだ。

11.3　LLVM のコード表現: LLVM IR

歴史的な経緯やコンテキストはさておいて、LLVM の内部に飛び込んでみよう。LLVM の設計でいちばん重要な部分が LLVM Intermediate Representation (IR) だ。これは、コンパイラ内でのコードの表現形式である。LLVM IR は中間レベルの解析や変換のために作られたもので、コンパイラの中のオプティマイザの部分で使う。いろんな目標があった。軽量な実行時最適化、関数や手続きをまたがった最適化、プログラム全体の解析、積極的な構造変換などだ。しかし、中でも一番重要だったのは、それ自体をファーストクラスの言語として定義し、きちんと考えた文法にすることだった。具体例として、シンプルな`.ll`ファイルを示す。

```
define i32 @add1(i32 %a, i32 %b) {
entry:
  %tmp1 = add i32 %a, %b
  ret i32 %tmp1
}

define i32 @add2(i32 %a, i32 %b) {
entry:
  %tmp1 = icmp eq i32 %a, 0
  br i1 %tmp1, label %done, label %recurse

recurse:
  %tmp2 = sub i32 %a, 1
  %tmp3 = add i32 %b, 1
  %tmp4 = call i32 @add2(i32 %tmp2, i32 %tmp3)
```

```
  ret i32 %tmp4

done:
  ret i32 %b
}
```

この LLVM IR は、次の C のコードに対応するものだ。このコードでは、二通りの方法で整数の足し算をしている。

```
unsigned add1(unsigned a, unsigned b) {
  return a+b;
}

// 非効率的な方法での足し算
unsigned add2(unsigned a, unsigned b) {
  if (a == 0) return b;
  return add2(a-1, b+1);
}
```

この例からわかるとおり、LLVM IR はローレベルな RISC 風の仮想命令セットである。実際の RISC 命令セットと同様に、add や subtract、compare、branch といったシンプルな命令のリニアなシーケンスに対応している。これらの命令は 3 アドレス形式で、いくつかの入力を受け取って、結果を別のレジスタに生成するというものだ。[5]LLVM IR はラベルをサポートしており、ちょっと見た限りでは風変わりなアセンブリ言語にも見える。

多くの RISC 命令セットとは異なり、LLVM はシンプルな型システムによる強い型付け (たとえば `i32` は 32 ビット整数、`i32**`は 32 ビット整数へのポインタのポインタ) があり、マシンの詳細は抽象化されている。たとえば、呼び出し規約の抽象化をしているのが `call` 命令と `ret` 命令で、これらに明示的に引数を指定する。マシンコードとの大きな違いがもうひとつあって、それは LLVM IR が名前付きレジスタの固定セットを使わないということだ。代わりに使うのは無限のテンポラリレジスタで、名前には%文字を使う。

言語としての実装にとどまらず、LLVM IR は実際には三通りの形式で定義されている。先に示したテキスト形式、オプティマイザがインメモリで扱うときのデータ構造、そして効率的かつ濃縮した形式でディスク上に格納するバイナリ形式の “bitcode” フォーマットだ。LLVM プロジェクトでは、ディスク上のフォーマットをテキストからバイナリに変換するツールも用意している。`llvm-as` はテキスト形式の`.ll` ファイルを bitcode 形式の`.bc` ファイルに変換し、`llvm-dis` はその逆で`.bc` ファイルから`.ll` ファイルに変換する。

コンパイラの中間表現は興味深いものだ。なぜならそれは、コンパイラのオプティマイザにとって “理想の世界” だからである。コンパイラのフロントエンドやバックエンドとは違い、オプティマイザは特定のソース言語やターゲットマシンの制約を受けない。一方、オプティマイザはその両方を適切に扱えないといけない。フロントエンドが生成しやすいような

[5]これは、2 アドレス形式 (X86 など、入力レジスタを破壊してしまう方式) や 1 アドレスマシン (オペランドだけを指定してアキュムレータを操作したり、スタックマシン上でスタックの先頭を操作したりするもの) とは対象的だ。

設計にしないといけないし、実際のターゲット上での最適化が十分できるほどの表現力も必要だ。

LLVM IR の最適化

最適化の様子を直感的につかむには、例を使って実際に動きを見るのがいいだろう。コンパイラの最適化にはいろんなやりかたがあるので、あらゆる問題に対応できるレシピを用意するのは難しい。とはいえ、ほとんどの最適化はこの三段階の構造になっている。

- 変換できそうなパターンを探す。
- その変換をしても安全か、そして動きが変わらないかを確かめる。
- 変換を施し、コードを更新する。

いちばんわかりやすい最適化は、数式上のパターンマッチングだ。たとえば、任意の整数 `X` について `X-X` は 0 になるし、`X-0` は `X`、`(X*2)-X` は `X` になる。これを実現するための最初の課題は、これらの式が LLVM IR 上でどのような表現になるのかを知ることだ。たとえば、こんなふうになる。

```
...
%example1 = sub i32 %a, %a
...
%example2 = sub i32 %b, 0
...
%tmp = mul i32 %c, 2
%example3 = sub i32 %tmp, %c
...
```

この種の「節穴」的な変換用に、LLVM では命令を単純化するインターフェイスを用意している。これをユーティリティとして使い、より上位レベルのさまざまな変換を行う。今回の例の変換は、`SimplifySubInst` 関数を使って次のようにできる。

```
// X - 0 -> X
if (match(Op1, m_Zero()))
  return Op0;

// X - X -> 0
if (Op0 == Op1)
  return Constant::getNullValue(Op0->getType());

// (X*2) - X -> X
if (match(Op0, m_Mul(m_Specific(Op1), m_ConstantInt<2>())))
  return Op1;

...

return 0;  // どれにもマッチしなかったときは、null を返して何も変換しないことを示す
```

このコードでは、Op0 と Op1 がそれぞれ整数減算命令の左右のオペランドに束縛される (これらは IEEE 浮動小数点にも適用できるとは限らないことに注意!)。LLVM は C++で実装されているが、C++のパターンマッチは (Objective Caml などの関数型言語と比べて) そんなに高機能ではない。しかし C++には非常に汎用的なテンプレートシステムがあり、それと同等の仕組みを実装できる。`match` 関数と `m_`関数は、宣言型のパターンマッチ操作を LLVM IR コードに対して行う。たとえば `m_Specific` 述語は、乗算の左辺が Op1 と等しい場合にだけマッチする。

これら三つのケースがどれもパターンマッチの対象となり、もし変換可能なら変換結果が返される。変換できなかった場合は null ポインタが返される。この関数 (`SimplifyInstruction`) の呼び出し元はディスパッチャで、命令のオペコードに応じてそれぞれのオペコード用のヘルパー関数に振り分ける。これが、各種最適化の場面で呼ばれる。シンプルなドライバの例を、次に示す。

```
for (BasicBlock::iterator I = BB->begin(), E = BB->end(); I != E; ++I)
  if (Value *V = SimplifyInstruction(I))
    I->replaceAllUsesWith(V);
```

このコードは単にブロック内の各命令をループして、単純化できるものがあるかどうかを調べる。もしできるなら (つまり `SimplifyInstruction` が非 null を返したら)、`replaceAllUsesWith` メソッドを使ってよりシンプルな形式の操作に置き換える。

11.4 LLVM での 3 フェーズ設計の実装

LLVM ベースのコンパイラにおけるフロントエンドの役割は、入力されたコードのパースと検証そしてエラー診断で、パース済みのコードを LLVM IR に変換する (場合によっては、AST を構築してからそれを LLVM IR に変換することもある)。この IR を、(オプションで) 解析や最適化に回してコードを改善させ、それからコードジェネレータに送ってネイティブのマシン語を生成する。その様子を図 11.3 に示す。3 フェーズ設計をそのままストレートに実装したものだが、この単純な説明の中に、LLVM のアーキテクチャが LLVM IR から受け継いだ威力や柔軟性が隠されている。

LLVM IR は完全なコード表現である

より具体的に言うと、LLVM IR はきちんと規定されていて、かつオプティマイザへの**唯一の**インターフェイスであるということだ。つまり、LLVM のフロントエンドを書くのに必要となる知識は、LLVM IR が何者かということとその動作原理そして期待する不変条件だけとなる。LLVM IR にはファーストクラスのテキスト表現形式があるので、LLVM IR をテキスト形式で出力するフロントエンドを作ってもかまわないし、それも十分理にかなっている。それを Unix のパイプに通してオプティマイザやコードジェネレータに渡せばいい。

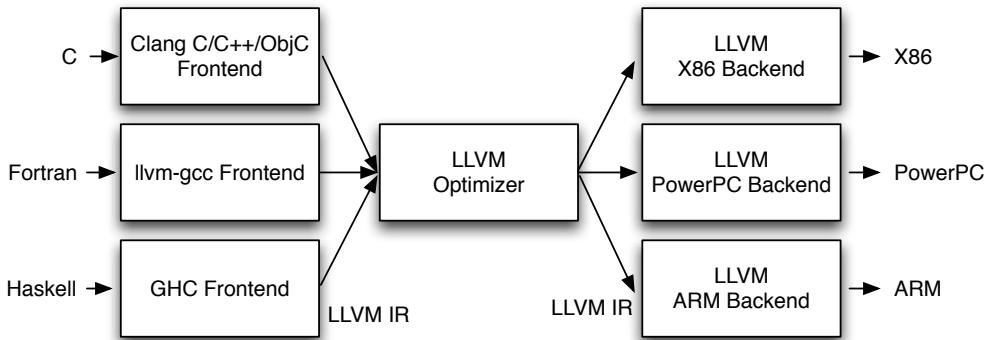



図 11.3: LLVM での 3 フェーズ設計の実装

驚くかもしれない。でもこれこそが LLVM の今までにない特徴となって、いろんなアプリケーションで幅広く使われるようになった理由にもなったのだ。大いに成功を収めて比較的きれいなアーキテクチャになっている GCC コンパイラでさえも、これは実現できなかった。GCC の中間表現である GIMPLE は、自己完結していなかったのだ。単純な例をあげると、GCC のコードジェネレータは DWARF 形式のデバッグ情報を発行するが、これはソースレベルの「ツリー」形式にまで達するものだ。GIMPLE 自身は「タプル」表現を使ってコード内の操作を表しているが、(少なくとも GCC 4.5 の時点では) オペランドの表現はソースレベルのツリー形式にさかのぼっている。

これが何を意味するかというと、GCC のフロントエンドを書きたければ、GIMPLE だけではなく GCC のツリーデータ構造も知らないといけないし、それを生成しないといけないということだ。GCC は、バックエンド側でも同様の問題を抱えている。つまり、バックエンドを書くためには RTL バックエンドについても多少は知る必要があるということだ。GCC には「自分のコードを表すすべて」を出力する方法がない。そして、GIMPLE(および、コードを表すためのそれに関連するデータ構造) をテキスト形式で読み書きする方法もない。その結果、GCC は比較的扱いにくくなり、フロントエンドの数も多くないということだ。

LLVM はライブラリ群である

LLVM IR そのものの設計は別として、それ以外でもっとも重要な LLVM の性質は、ライブラリ群として作られているという点だ。GCC みたいなモノリシックなコマンドラインのコンパイラ、そして JVM あるいは.NET 仮想マシンのような不透明な仮想マシンとはこの点が異なる。LLVM は基盤であり、有用なコンパイラ技術を集めたものだ。これらを使って、さまざまな固有の問題 (C コンパイラを作ったり、特殊効果パイプライン内でのオプティマイザを作ったりなど) に対応できる。これは最も強力な機能の一つであるにも関わらず、最も過小評価されている点でもある。

オプティマイザの設計を例にとって考えてみよう。オプティマイザはLLVM IR形式の入力を読んでいろいろかみ砕き、同じくLLVM IR形式を出力する。出力のほうが実行速度が速くなっていることだろう。LLVMのオプティマイザは、その他多くのコンパイラの場合と同じく、いくつかの最適化パスのパイプラインで構成されている。各パスが、入力を受け取って何らかの操作をする。よくあるパスの例は、インライナー (関数の本体を呼び出し側に移動させる) や式の再構成、ループ内で不変なコードの移動などだ。最適化のレベルに応じて、実行するパスが変わる。-O0 (最適化しない) の場合はClangコンパイラはどのパスも実行しないが、-O3の場合はコンパイラ内で67のパスを実行する (LLVM 2.8時点での値)。

LLVMのパスはC++のクラスとして書かれており、(間接的に)Passクラスを継承している。ほとんどのパスは単一の.cppファイルにまとめられており、Passクラスのサブクラスは無名名前空間で定義されている (つまり、定義しているファイル内で完全に閉じたものになる)。パスを使いやすくするためには、そのファイルの外部のコードからも取得できなければいけない。そこで、パスを作るための関数をファイルから公開する。若干単純にした具体例を示す。[6]

```
namespace {
  class Hello : public FunctionPass {
  public:
    // 最適化対象の LLVM IR 内にある関数の名前を表示する
    virtual bool runOnFunction(Function &F) {
      cerr << "Hello: " << F.getName() << "\n";
      return false;
    }
  };
}

FunctionPass *createHelloPass() { return new Hello(); }
```

既に説明したとおり、LLVMのオプティマイザは何十ものパスを提供しており、それぞれ似たスタイルで書かれている。これらのパスがコンパイルされてひとつあるいは複数の.oファイルになり、それをアーカイブライブラリ (Unixシステムの場合は.aファイル) に組み込む。これらのライブラリが解析や変換などのあらゆる機能を提供し、パスを可能な限り疎結合にする。つまり、各パスは基本的にそれ単体で動くものとされており、もし他の解析などに依存している場合は他のパスとの依存関係を明示する。実行する一連のパスを指定すると、LLVMのPassManagerはその依存情報を使って依存関係を見たし、パスの実行を最適化する。

ライブラリや機能の抽象化はすばらしいが、それだけで万事解決というわけではない。興味深い例を考えてみよう。誰かがコンパイラの技術を活用する新しいツールを作ろうとしたとする。たとえば画像処理言語用のJITコンパイラなどだ。このJITコンパイラを実装する人は、いろんな制約を考慮しないといけない。たとえば、その画像処理言語がコンパイル時のレイテンシーにとても厳しいものかもしれない。そして、その言語に独特の特徴があって、パフォーマンス上の理由で最適化できなかったりするかもしれない。

[6]詳細は、マニュアルの *Writing an LLVM Pass* (http://llvm.org/docs/WritingAnLLVMPass.html) を参照のこと。

LLVMのオプティマイザはライブラリベースの設計になっているので、実装する人がどのパスをどの順番で実行するか取捨選択できる。それは、画像処理のドメインでも理にかなっているだろう。もしたった一つの大きな関数にすべてまとまっていたら、その中から必要な部分だけをインラインで切り出すという作業に無駄な時間を費やすことになる。もしポインタがほとんどなければ、エイリアスの解析やメモリの最適化はそれほど面倒ではない。でも、いくら最善を尽くしたところで、LLVMが最適化の問題を魔法のように解決してくれるわけじゃない！パスのシステムはモジュール化されており、PassManager自身はパスの内部について関知しないので、実装者はその言語専用のパスでLLVMオプティマイザの不備をカバーする必要はない。また、明示的にその言語専用の最適化をする必要もない。図11.4は、その一例として、架空の画像処理システムXYZを示したものだ。

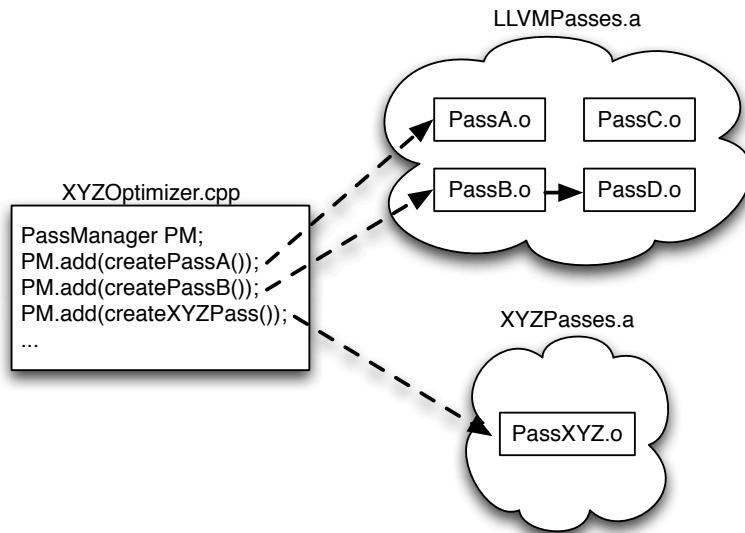



図11.4: LLVMを使った架空のXYZシステム

利用する最適化のセットの選択(と、コードジェネレータについてのそれと同様の選択)が終わったら、画像処理コンパイラを実行形式にするか動的ライブラリとして組み込む。LLVMの最適化パスへの唯一の参照は`.o`ファイルで定義されたシンプルな`create`関数であり、オプティマイザ自体はアーカイブライブラリ`.a`に存在するので、**実際に利用する**最適化パスだけがアプリケーションにリンクされる。LLVMオプティマイザ全体がリンクされるわけではない。先ほどの例ではPassAとPassBへの参照があるので、これらがリンクされる。また、PassBがPassDを使って何らかの解析をしているので、PassDも組み込まれる。しかし、PassC(そしてその他の最適化パス)は使われていないので、これらは画像処理アプリケーションにはリンクされない。

ここで、LLVMがライブラリベースであるという特徴が威力を発揮する。このアプローチで設計していたおかげで、LLVMはいろんな可能性を手に入れた。ごく一部の人たちだけに

しか役立たないようなものもあるが、それがその他大勢の人 (ただ単にシンプルに使いたいだけの人) たちに害を及ぼすことはない。それとは対照的に、昔ながらのコンパイラのオプティマイザは大部分のコードが密結合しており、その一部だけを取り出して高速化するのはかなり難しい。LLVM なら、ある特定のオプティマイザを理解するのにシステム全体に関する知識は不要だ。

このライブラリベースの設計のせいで、多くの人が LLVM について誤解しているところもある。LLVM のライブラリにはいろんな機能があるが、それ単体では実際には**何もしない**。ライブラリをいかに組み合わせて機能を実現するかは、ライブラリを使うクライアント (Clang C コンパイラなど) の設計に依存するということだ。注意深くレイヤー化され、分割され、それぞれの機能に焦点を合わせていることで、LLVM のオプティマイザはいろんな場面でいろんなアプリケーションから使えるようになった。また、LLVM が JIT コンパイル機能を提供しているからといって、すべてのクライアントがその機能を使っているというわけでもない。

11.5　再利用性を考慮した LLVM コードジェネレータの設計

LLVM のコードジェネレータの役割は、LLVM IR を各ターゲット向けのマシンコードに変換することだ。指定されたターゲット向けに向けて、いちばんいいコードを作るというのもコードジェネレータの仕事だ。理想を言えば、コードジェネレータは個々のターゲットに特化したカスタムコードになるべきだろう。しかしその一方で、各ターゲット向けのコードジェネレータが解こうとしている問題は、どれも似たようなものだ。たとえば、値をレジスタに割り当てる必要があるというのはどのターゲットでも同じことだ。レジスタファイルはターゲットごとに異なるだろうが、そのときに使うアルゴリズムは可能な限り共有しておくべきだろう。

オプティマイザのときと同様に、LLVM のコードジェネレータも、コード生成に関する問題を個別のパス (命令の選択、レジスタの割り当て、スケジューリング、コード配置の最適化、アセンブリコードの発行など) に分割して、さまざまなパスをデフォルトで使えるよう組み込んでいる。各ターゲット用のコードジェネレータを作ろうとする人はデフォルトのパスの中から好きなものを使えるし、それをオーバーライドしてそのターゲット専用にカスタマイズしてしまってもいい。たとえば x86 用のバックエンドはレジスタ数を節約するスケジューラを使っている。これは x86 のレジスタ数が限られているからだが、一方 PowerPC 用のバックエンドではレイテンシーを最適化したスケジューラを使っている。レジスタには余裕があるからだ。x86 バックエンドではカスタムパスを使って x87 浮動小数点スタックを扱っており、ARM バックエンドでもカスタムパスを使って定数プールを関数内で必要に応じて配置する。この柔軟性のおかげで、バックエンドの作者はそのターゲット用のコードジェネレータをゼロから書きあげる必要がなくなるというわけだ。

LLVM ターゲット記述ファイル

この「いろんなものを組み合わせる」手法のおかげで、バックエンドの作者はそのアーキテクチャに合わせた選択ができるようになる。そして、ターゲットをまたがってコードを再利用できるようにもなる。しかし、その一方で問題も出てくる。共有コンポーネントは、何らかの汎用的な方法でターゲット固有の特性を判断できないといけない。たとえば、レジスタアロケータを共有する場合は、ターゲットごとのレジスタファイルの場所や命令とレジスタオペランドとの間の制約などについて知っておく必要がある。LLVM は、この問題を解決するために、ターゲットごとの記述をドメイン特化言語 (`.td` ファイルの集まり) で提供するようにした。この記述は tblgen で生成する。x86 ターゲット用の (簡略化した) ビルド手順を図 11.5 に示す。

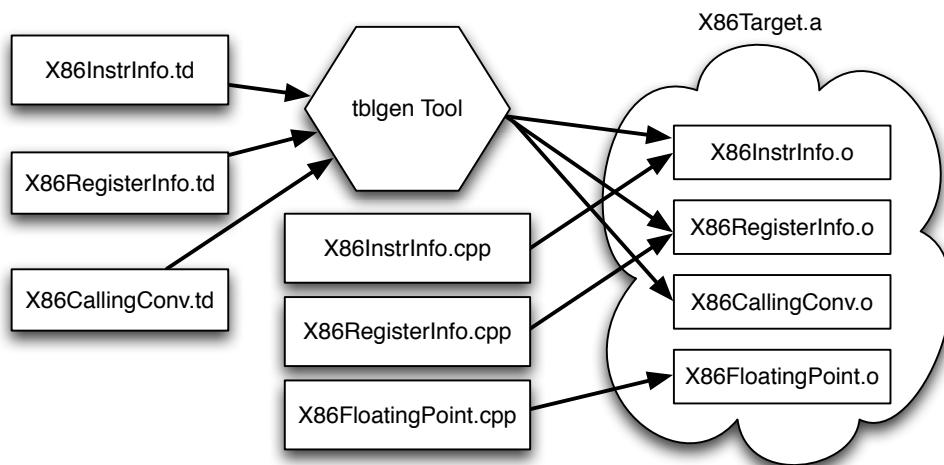



図 11.5: 簡略化した x86 ターゲット定義

`.td` ファイルがサポートするさまざまなサブシステムを使えば、バックエンドの作者は各ターゲット用にいろんなピースを作れるようになる。たとえば x86 バックエンドで定義しているレジスタクラスは、“GR32”(`.td` ファイル内では、ターゲット固有の定義は大文字になる) という名前で 32 ビットレジスタをすべて保持している。

```
def GR32 : RegisterClass<[i32], 32,
  [EAX, ECX, EDX, ESI, EDI, EBX, EBP, ESP,
   R8D, R9D, R10D, R11D, R14D, R15D, R12D, R13D]> { ... }
```

ここで定義しているのは、このクラスに属するレジスタに 32 ビット整数値 (“i32”) を保持できること、32 ビット単位で整列されること、16 種類のレジスタがある (その定義は `.td` ファイル中のどこかにある) こと、そして代入の順序などのその他の情報だ。これを定義しておけば、各命令から参照でき、オペランドとして使えるようになる。たとえば、「32 ビットレジスタの補集合をとる」という命令の定義はこのようになる。

```
let Constraints = "$src = $dst" in
def NOT32r : I<0xF7, MRM2r,
               (outs GR32:$dst), (ins GR32:$src),
               "not{l}\t$dst",
               [(set GR32:$dst, (not GR32:$src))]>;
```

この定義では、まずNOT32rが命令であるということを示し(tblgenクラスIを使っている)、続けて符号化方式の情報(0xF7, MRM2r)を指定し、さらに出力用の32ビットレジスタ$dstと入力用の32ビットレジスタ$src(先ほどの定義のレジスタクラスGR32で、このオペランドに使えるレジスタがわかる)を指定する。さらに、その命令のアセンブリ構文({}を使ってAT&TとIntelの両方を扱う)とその命令の効果を指定し、マッチするパターンを最後の行で指定する。最初の行の“let”制約は、入出力用のレジスタをそれぞれ同じ物理レジスタ上に確保する必要があることをレジスタアロケータに指定する。

この定義には命令に関する説明がぎっしり詰まっており、LLVMのコードは(tblgenを使って)ここから多くの情報を引き出せる。命令の選択をするにはこの定義だけで十分で、入力のIR形式のコードに対するパターンマッチングでこの命令を作る。また、レジスタアロケータにもその処理方法を伝えられるし、命令からバイトコードへの符号化や復号もできる。命令をテキスト形式で表示するにもこの定義だけで十分だ。これらを使えば、x86用のバックエンドで(“gas” GNUアセンブラの一時的な代替として)スタンドアロンのx86アセンブラ形式を生成させたりターゲット記述からのディスアセンブラをサポートしたりもできる。またJIT用に命令を符号化することもできる。

便利な機能を提供できることだけに限らず、複数の情報をたったひとつの「正解」から生成できるということも利点のひとつだ。この方式にしておけば、アセンブラとディスアセンブラが構文やバイナリ符号化方式などで食い違うことがまずなくなる。また、ターゲット記述のテストもしやすい。命令の符号化を単体テストするときに、コードジェネレータ全体を巻き込まなくても済むようになる。

ターゲットに関する情報は可能な限り.tdファイルに取り込んで、宣言型の書式でまとめられるようにしてきた。しかし、すべてを取り込めたわけではない。そのかわりに、バックエンドの作者にちょっとしたC++のコードを書いてもらうようにしている。各種サポートルーチンや、そのターゲットにだけ必要になるようなパス(x87浮動小数点スタックを扱うX86FloatingPoint.cppなど)の実装は、そちらで扱う。LLVMが新たなターゲットに対応して成長するにつれて、.tdファイルで表現しきれるターゲットの量を増やすことがより重要になる。そのために、私たちは.tdファイルの表現力を向上させ続ける。表現力が上がれば上がるほど、LLVM用のターゲットがより書きやすくなるだろう。

11.6 モジュラー設計がもたらす興味深い可能性

全体的に洗練されているという点以外にも、モジュラー設計の利点はある。LLVMのライブラリを使うクライアントに、いくつかの興味深い可能性をもたらしてくれるのだ。そもそ

もは LLVM 自体の機能によるものだが、クライアント側で、それをどう使うかという**ポリシー**を決められるというのが大きい。

各フェーズを動かすタイミングを選ぶ

先述のとおり、LLVM IR はバイナリ形式 (LLVM ビットコード) との間でのフォーマット変換ができる。LLVM IR 自体が自己完結しているうえにフォーマット変換がロスレスな処理であることから、コンパイル処理の一部だけを行ってその状況をディスクに保存し、また違うときにその続きを実行するなどということもできる。この昨日を使えば、リンク時の最適化やインストール時の最適化なども実現できる。どちらも、「コンパイル時」のコード生成を遅らせるものだ。

リンク時の最適化 (Link-Time Optimization: LTO) は、コンパイラが伝統的にもつ問題に対応するものだ。その問題とは、一度に一つの変換ユニット (e.g., a `.c` file with all its headers) しか見ないために、ファイルをまたがる最適化 (インライン化など) ができないというものだ。LLVM のコンパイラ (Clang など) は、これに対応したコマンドラインオプション`-flto` あるいは`-O4` を用意している。このオプションは、ネイティブなオブジェクトファイルではなく`.o` ファイルに LLVM ビットコードを書き出すよう指示する。そして、コードの生成はリンク時まで先延ばしにする。その様子を図 11.6 に示す。

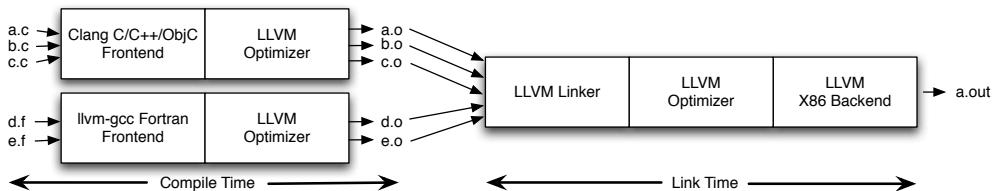



図 11.6: リンク時の最適化

OS によって詳細は異なるが、重要なのは、LLVM ビットコードがネイティブオブジェクトファイルではなく`.o` ファイルにあるというのをリンカが気づくということだ。リンカがそれを発見したら、すべてのビットコードをメモリに読み込む。そしてそれをリンクし、とりまとめた結果に対して LLVM オプティマイザを実行する。オプティマイザはかなり大きなコード片を扱えるようになり、インライン化や定数の伝搬、より積極的なデッドコードの破棄などの最適化をファイルをまたがって行えるようになる。最近のコンパイラの多くは LTO をサポートしているが、そのほとんど (GCC や Open64、Intel compiler など) は、コストのかかる低速なシリアライズ処理を使って実現している。LLVM の場合は、システムの設計からごく自然に LTO が実現でき、異なるソース言語の組み合わせでも利用できる (他のコンパイラでは無理なことだ)。これもまた、IR が本当の意味でソース言語に中立だからこそ実現できたことだ。

インストール時の最適化とは、コードの生成をリンク時よりもさらに先延ばしにして、インストールのときに行うというものだ。その様子を図 11.7 に示す。インストール時というのは (危機に組み込んだり、ダウンロードしたり、モバイルデバイスにアップロードしたりなど) いろいろ興味深いタイミングだ。というのも、導入しようとしているデバイスの詳細が判明するのがまさにこのときだからである。x86 ファミリーを例に取ると、いろんなチップが存在するしそれぞれ特性は違う。命令の選択やスケジューリングなどのコード生成がらみの処理を先送りすることで、最終的にアプリケーションを実行する環境のハードウェアにより特化した選択ができるようになる。

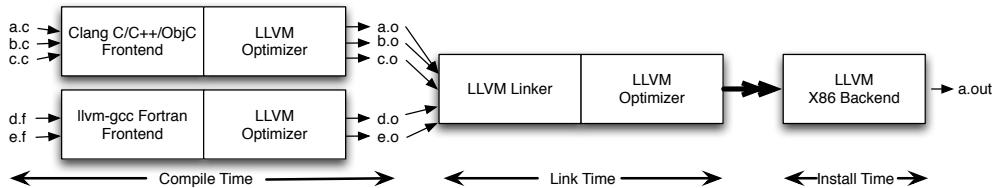



図 11.7: インストール時の最適化

オプティマイザのユニットテスト

コンパイラは非常に込み入ったものであり、高い品質が要求される。そのため、テストが重要になる。たとえば、オプティマイザでクラッシュを引き起こすようなバグを修正したとしたら、同じバグが再発しないようにリグレッションテストも追加しないといけない。リグレッションテストを書く昔ながらの手法は、コンパイラを通す (たとえば) .c ファイルを書いて、コンパイラがクラッシュしないことを確かめるテストハーネスを用意するというものだ。この手法は、たとえば GCC のテストスイートでも使われている。

この手法の問題は、コンパイラがさまざまなサブシステムで構成されていて、オプティマイザにもさまざまなパスがあるということだ。そのすべてが、バグのあった箇所への入力となるコードに対して何かの変更を加える可能性がある。フロントエンドやオプティマイザの初期段階で何かを変更すると、これまでのテストケースが動かなくなりやすい。

LLVM IR のテキスト形式とモジュール式のオプティマイザを使うことで、LLVM のテストスイートはリグレッションテストに注力できるようになる。LLVM IR をディスクから読み込んで、特定の最適化パスを実行し、期待される挙動を検証できるのだ。クラッシュするかしないかというレベルではなく、もっと入り組んだ振る舞いをテストしたいこともある。最適化が実際に行われているのかというテストなどだ。ここで、シンプルなテストケースの例を示す。この例は、定数の伝搬の最適化が add 命令に対して働いているかどうかのテストだ。

```
; RUN: opt < %s -constprop -S | FileCheck %s
define i32 @test() {
  %A = add i32 4, 5
  ret i32 %A
```

```
  ; CHECK: @test()
  ; CHECK: ret i32 9
}
```

RUN の行では、実行するコマンドを指定する。この例の場合は、コマンドラインツール opt と FileCheck だ。opt は LLVM パスマネージャーのシンプルなラッパーで、すべての標準パスをリンクして (そして、その他のパスを含むプラグインも動的に読み込めて)、それをコマンドラインに公開する。FileCheck は、標準入力に与えられた内容が一連の CHECK ディレクティブにマッチするかどうかを確かめる。今回の場合は、constprop パスが 4 と 5 の add を 9 にまとめるかどうかを調べている。

何てこともないごく当たり前の例に見えるかもしれないが、同じことを.c ファイルを書いてテストしようとすると非常に難しい。フロントエンドが定数のたたみ込みをパース時に済ませてしまうことが多いので、わざわざそれを先延ばしにして、オプティマイザのパスで処理させるコードを書くのはとても難しいし不安定になる。LLVM の場合は、LLVM IR をテキスト形式で読み込んで特定のパスに送り、その結果を別のテキストファイルに書き出すだけだ。リグレッションテストについても機能テストについても、極めて直感的に実現できる。

BugPoint による、自動的なテストケースの単純化

コンパイラ、あるいは LLVM のライブラリを使う他のクライアントに何らかのバグが見つかったとして、それを修正するためにまず最初にすることは、問題を再現するためのテストケースを書くことだ。テストケースさえ書けてしまえば、問題を再現する必要最小限のサンプルが得られたことになる。そして、問題の発生源を LLVM の特定のパーツ、たとえば特定の最適化パスなどに絞り込めることにもなる。読者もいずれはその方法を身につけることになるだろうが、その手順はつまらない手作業になる。特に、コンパイラが生成するコードが何かおかしいけれどもクラッシュに至るほどではない、という場合の作業が難しい。

LLVM の BugPoint[7] というツールは、IR のシリアライズ機能と LLVM のモジュラー構成を活用してこの手順を自動化する。たとえば、入力として.ll ファイルか.bc ファイルを与え、さらにオプティマイザのクラッシュを引き起こす最適化パスのリストを渡すと、BugPoint が入力を単純化して小さなテストケースを作り、どのパスが問題なのかを特定する。それから単純化したテストケースを出力し、opt コマンドを使って障害を再現する。このツールが問題の原因を特定するときに使うのが “Delta Debugging” と似たテクニックで、これを使って入力と最適化パスのリストを減らしていく。BugPoint は LLVM IR の構造を知っているので、オプティマイザが理解できないような無効な IR を生成したりはしない。標準のコマンドラインツール “delta” では、時にそんなことで時間を無駄にすることがある。

もう少し複雑なコンパイルミスの場合は、入力とコードジェネレータの情報、そして実行形式に渡すコマンドラインや参考出力も指定できる。BugPoint は、まずその問題がオプティマイザによるものなのかコードジェネレータによるものなのかを判断する。それからテスト

[7] http://llvm.org/docs/Bugpoint.html

ケースを繰り返し二分割して、一方を「正常に動く」コンポーネント、もう一方を「バグがある」コンポーネントとして振り分ける。分割を繰り返していくことで「バグのある」コンポーネントに送られる部分が減っていき、テストケースの単純化につなげる。

BugPoint は非常にシンプルなツールで、LLVM のテストケースを作る時間を数え切れないほど節約してくれた。オープンソースのコンパイラの中でこれほど強力なツールを持ち合わせているものは、他にないだろう。うまく作られた LLVM の中間表現があってこそ実現できたツールだからである。とはいうものの、BugPoint は決してパーフェクトなツールではなく、書き直せばもっといいものになるだろう。2002 年に戻って振り返ってみても、このツールに手を入れる機会は限られていた。誰かが奇妙なバグに遭遇して既存のツールではうまく処理できなかったときに、このツールに手を入れるというものだ。そんな具合い徐々に成長し、一貫した設計や作者もなしに新たな機能 (JIT デバッグなど) が追加されてきた。

11.7　ふりかえりと今後の方針

モジュール方式による LLVM の設計は、本書で扱ったようなメリットを当初から意図していたものではない。あくまでも自衛のためにとった方式だったのだ。最初から最高のものができあがることなんてあり得ない。たとえばパスのパイプラインをモジュール式にしたのも、パスごとに分離しておけば、よりよい実装ができたときに取り替えやすくなるからだった[8]。

もうひとつ、LLVM を軽量なままに保ち続けている大きな特徴がある (ライブラリを使うクライアント側から見ると賛否両論がある)。それは、過去の決断も積極的に見直して、過去との互換性を気にせずに API を大きく変更していくということだ。たとえば LLVM IR 自体に大幅な変更を加えるには、すべての最適化パスの変更が必要になる。そしてそれは、C++ の API にも大きな影響を及ぼす。LLVM では過去に何度かそういうことがあった。クライアント側にとっては辛かっただろうが、今後の開発を順調に進めていくためにはそうすべきだった。外部のクライアント (そして他言語のバインディング) の手間を減らすために、主要 API(完全に安定したもの) の多くには C のラッパーを提供している。また、新しいバージョンの LLVM でも、旧バージョンの`.ll` や`.bc` を読めるようにし続けている。

今後の展望としては、LLVM をもっとモジュール化して、サブセットを作りやすいようにし続けていきたい。コードジェネレータなどは、まだ一枚岩になりすぎている。現時点では、LLVM のコードジェネレータの一部の機能だけを利用するなどということはできない。JIT は使いたいけれどもインラインアセンブラや例外処理はいらないし、デバッグ情報の生成機能もいらないという場合は、不要な機能のサポートをリンクせずにコードジェネレータをビルドしないといけない。また、オプティマイザやコードジェネレータが作るコードの品質を上げることにも常に取り組み続けている。さらに、IR で新たな言語やターゲットをサポートしやすくなるような機能を追加したり、より上位レベルで言語固有の最適化ができるようにすることも考えている。

[8]LLVM のサブシステムの中で、もう完璧だから書き直す余地がないなんてものは存在しない。

LLVM プロジェクトは、いろんな方法で成長し続け、改良されていく。さまざまなプロジェクトで、いろんな方法で LLVM が活用されているのを見るのはとてもうれしいし、私たちが思ってもいなかったような斬新な使い方に驚かされることも多い。新しいデバッガである LLDB は、その最たる例だ。これは、Clang の C/C++/Objective-C パーサーを使って式を解析し、LLVM JIT を使ってそれをターゲットのコードに変換し、LLVM のディスアセンブラを使い、さらに LLVM のバックエンドを使って呼び出し規約などを処理する。これらのすばらしいコードを再利用できるようにしたおかげで、デバッガの開発者はデバッグそのもののロジックを書くことだけに集中できるようになった。C++のパーサーの再実装などせずに済んだということだ。

ここまではなんとかうまくやってきたが、まだやり残したことは多い。さらに、今後 LLVM が年を重ねるにつれて、軽快さが失われて硬直化してしまうというリスクもある。この問題には魔法のような解決策があるわけではない。でも、新しい問題領域を公開し続けていることや過去の決断を躊躇せず再考していること、さらに再設計で過去のコードを捨てられるようにしていることなどが、すこしでもその対策になって欲しいものだ。結局のところ、パーフェクトな存在を目指すのではなく、常に向上し続けることが大切なのだ。

第12章

Mercurial

Dirkjan Ochtman

Mercurial はモダンな分散型バージョン管理システム (VCS) で、大半は Python で書かれている。ただ、ごく一部、パフォーマンスのために C で書いているところもある。本章では、Mercurial のアルゴリズムやデータ構造を設計したときのいくつかの判断について説明する。まず最初に、バージョン管理システムの歴史を簡単に振り返っておこう。これが、本章を読み進めるための前提知識となる。

12.1 バージョン管理システムの簡単な歴史

本章で主に扱うのは Mercurial のアーキテクチャについてである。しかし、その概念の多くは他のバージョン管理システムと共通している。Mercurial についての議論を実のあるものにするために、まずはさまざまなバージョン管理システムの概念や動作に名前をつけるところからはじめよう。さらに、それらすべてを見渡すために、この世界の歴史についても簡単に説明する。

バージョン管理システムが作られたのは、複数人によるソフトウェアシステムの開発作業を支援するためだった。ソースの完全なコピーをやりとりして変更履歴を各自で管理させるなどということをしなくて済むように、と作られたのだ。ここで、単なるソフトウェアのソースコードだけではなく、任意のファイルツリーに一般化して考えよう。バージョン管理システムの主要機能のひとつは、変更をツリーに渡すことだ。基本的な流れは、このようになる。

1. 最新のファイルツリーを、どこか別の場所から取得する
2. このバージョンのツリー上で、何らかの変更を加える
3. 変更内容をどこかに公開し、他の人がそれを取得できるようにする

最初の操作、つまりファイルツリーをローカルに取得する作業のことを**チェックアウト**と呼ぶ。データの取得元であり、かつ変更点の公開先でもある格納場所のことは**リポジトリ**と呼び、チェックアウトしたもののことは**作業ディレクトリ**あるいは**作業ツリー**、**作業コピー**な

どと呼ぶ。作業コピーの内容をリポジトリ上の最新の状態に更新することは、単に**更新**と呼ぶ。このとき、場合によっては**マージ**が必要になることもある。マージとは、同一ファイル上での別のユーザーによる変更をとりまとめることだ。diff コマンドを使うと、ツリーあるいはファイルについて複数のリビジョン間での変更点を確認できる。最もよくある使いかたは、作業コピー上にあるローカルの (まだ公開していない) 変更を確認することだ。変更を公開するには *commit* コマンドを実行する。これは、作業ディレクトリ上での変更をリポジトリに記録する。

中央集中型バージョン管理

バージョン管理システムの元祖は Source Code Control System だ。略して SCCS と呼ばれるこのシステムが最初に登場したのは 1975 年のことだった。差分をひとつのファイルに記録するという方式でバージョンを管理しており、各バージョンのコピーを取っておくよりも効率的だった。その変更を他人に公開する仕組みは用意されていなかった。その後 1982 年に登場した Revision Control System(略して RCS) は、さらに進化しており、SCCS の代替として使えるフリーソフトウェアだった (RCS は、今でも GNU プロジェクトが保守を続けている)。

RCS に続いて登場したのが CVS。これは Concurrent Versioning System の略で、1986 年にリリースされた。当初は、RCS のリビジョンファイルを操作するためのスクリプト群という形式だった。CVS における大きなイノベーションは、複数のユーザーによる同時編集と、その後での編集のマージという概念だった。ここで、編集時の衝突という概念も登場する。開発者が何らかのファイルの新しいバージョンをコミットするためには、手元のファイルをリポジトリ内の最新版に基づいたものにしておく必要があった。リポジトリ内のファイルと作業ディレクトリ上のファイルの両方で (同じ行の) 変更があった場合は、両者の変更の衝突を解消しなければならなかった。

CVS は、**ブランチ**や**タグ**という概念を初めて取り入れたシステムでもある。ブランチのおかげで別々の作業を並行して行えるようになり、タグのおかげでスナップショットに名前をつけて容易に参照できるようになった。CVS の差分のやりとりは、当初は共有ファイルシステム上のリポジトリを使って行っていた。その後、CVS もクライアントサーバー型のアーキテクチャを採用し、(インターネットのような) 大規模ネットワーク上でも使えるようになった。

2000 年に、3 人の開発者が新しい VCS の開発をはじめた。それは Subversion と名付けられ、CVS の大きな問題点を修正することを目標として開発を進めた。最も重要だったのは、Subversion がツリー全体を一括して扱うことだった。つまり、リビジョン間での変更はアトミックであり、一貫性があってそれぞれ隔離されており、永続するということだ。Subversion の作業コピーには、作業ディレクトリのリビジョンをチェックアウトした当初の状態が残っている。つまり、よくある diff 操作 (ローカルのツリーをチェックアウトしたときの状態と比べる作業) がローカルで高速に実行できるということだ。

Subversion の興味深い概念のひとつが、タグやブランチもプロジェクトのツリーの一部とみなしたことである。Subversion のプロジェクトは、`tags`、`branches` そして `trunk` の三つ

のエリアに分かれていることが多い。この設計は、バージョン管理システムに不慣れなユーザーにとっては非常にわかりやすかった。しかし、この設計による柔軟性は、変換ツールを作る側にとってはさまざまな問題のもととなった。他のシステムでは、タグやブランチはそれ専用の構造で表すことが多かったからである。

ここまで取り上げたすべてのシステムは、いわゆる**中央集中型**である。CVS 以降のツールでは変更点をやりとりする方法を知っているが、それは他のコンピュータが変更の履歴を保持しているということに依存している。**分散型**のバージョン管理システムは、リポジトリの履歴のすべて (あるいは大半) を各コンピュータに保持しており、そのリポジトリの作業ディレクトリも保持している。

分散型バージョン管理

Subversion は CVS よりもずっときれいな実装だったが、それでも弱点はいくつか残っていた。たとえば、これは中央集中型のバージョン管理システムに共通する問題だが、変更をコミットすると同時に事実上その変更が公開されてしまうという点もそのひとつだ。リポジトリの履歴が中央で一元管理されている以上、どうしてもそうなってしまう。これはつまり、ネットワークに接続できない環境ではコミットが不可能であるということを意味する。第二に、中央集中型のシステムでリポジトリにアクセスする際には、最低一度はネットワーク上のやりとりが発生する。そのため、分散型システムの場合のローカルアクセスに比べて、比較的動作が遅くなってしまう。第三に、これまで取り上げたシステムは、どれもマージが下手である (中にはいくらかましになったものもあるが)。大規模なグループで並行作業をしていると、新しいバージョンがバージョン管理システムのどの変更を含むものかを把握することが重要となる。とりこぼしがないかを確認し、その後のマージをうまく行うためにこの情報を使うことが必要となる。第四に、昔ながらの VCS が求める中央管理は、時に人工的に見えることもある。統合のための場所を一カ所用意することになる。分散型の VCS を支持する人たちは、分散型のシステムのほうがより有機的な組織に対応できると言う。各開発者による変更のプッシュや統合を、必要に応じてその場で行えるというわけだ。

これらの問題を解決しようと、新たなツールがいろいろ登場した。私の周囲の世界 (オープンソースの世界) を見る限り、2011 年の時点での三大巨頭は Git と Mercurial そして Bazaar である。Git と Mercurial は、どちらも 2005 年に開発が始まった。Linux カーネルの開発者が、プロプライエタリな BitKeeper を使わないようにすると決断したころである。どちらのツールも Linux カーネルの開発者が最初に作り始めた (Git は Linus Torvalds、そして Mercurial は Matt Mackall だ)。何万ものファイル上で繰り広げられる何十万ものチェンジセット (そう、たとえば Linux カーネルみたいなもの) もきちんと扱えるように作られている。Matt も Linus も、Monotone VCS の影響を強く受けていた。Bazaar の開発はそれとは別に進んでいたが、Git や Mercurial と同時期に広く使われるようになった。Canonical がすべてのプロジェクトを Bazaar に移行したのがちょうどこの時期である。

分散型バージョン管理システムの構築にはいろいろ困難もあった。その多くは、分散型システムという性質によるものだった。一例を挙げよう。中央集中型のシステムにおけるソース管理サーバーは常に本流の履歴を提供できるが、分散型の VCS にはそもそも「本流の」履歴など存在しない。チェンジセットは各地で並行にコミットできるので、特定のリポジトリ上で一時的にリビジョンを並べることは不可能だ。

この問題に対して一般的に広く受け入れられている解決策は、チェンジセットを一列に並べるのではなくチェンジセットの無閉路有向グラフ (directed acyclic graph: DAG) を使うという方法だ (図 12.1)。つまり、新たにコミットしたチェンジセットは別のリビジョンの子リビジョンとなり、自分自身あるいは自分の子孫の子リビジョンになることはできないという構造である。この考え方を採用するために、三種類の特殊なリビジョンを用意した。親を持たないリビジョンである *root* **リビジョン** (ひとつのリポジトリに複数の root があってもかまわない)、複数の親を持つ**マージリビジョン**、そして子を持たない *head* **リビジョン**である。各リポジトリは、まず空の root リビジョンがひとつだけある状態から始まる。それを起点としてチェンジセットが連なり、最終的にひとつあるいは複数の head を持つことになる。二人のユーザーがそれぞれ別のコミットをしたときに、一方がもう一方のコミットを取り込みたくなったとしよう。そんな場合は、別のユーザーの変更を新しいリビジョンに明示的にマージしなければならない。この新しいリビジョンを、マージリビジョンとしてコミットする。

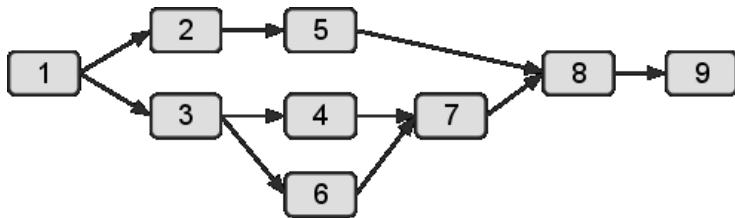



図 12.1: 各リビジョンの無閉路有向グラフ

DAG モデルを使うと、中央集中型のバージョン管理システムでは解決が困難な問題を解決する手助けになる。つまり、マージリビジョンを使えば、新たに DAG にマージされたブランチの情報を記録できるということだ。その結果としてできあがるグラフは並行するブランチ全体の大きなまとまりもうまく表せており、それをより小さなグループにマージしたり、最終的に「本流」ブランチにまとめたりもできる。

この手法を使うには、バージョン管理システムがチェンジセット間の親子関係を保持している必要がある。これを実現するため、チェンジセットのデータをやりとりする際には自分の親のチェンジセットを知っている状態にしておくことになる。当然、そのためにはチェンジセットに何らかの識別子が必要となる。システムによっては UUID やそれに類する仕組みを使った識別子を用意しているものもあるが、Git や Mercurial の場合はチェンジセットの内容の SHA1 ハッシュを使っている。この方式を使うと、チェンジセットの ID を使ってチェンジセットの中身が改ざんされていないことを検証できるという利点もある。実際、親の情報

もハッシュされたデータに含んでいるので、どのリビジョンからであってもそのハッシュだけで履歴の正当性を検証できる。コミットの作者名やコミットメッセージ、タイムスタンプ、その他のメタデータは、新しいリビジョンの実際の内容をハッシュするのと同様にハッシュされる。つまり、メタデータも同様に検証できるということだ。また、タイムスタンプはコミット時に記録されるので、任意のリポジトリ内でリニアに進む必要もない。

これまでに中央集中型の VCS しか使ってこなかった人たちにとっては、こういった考え方はなかなかなじめないものだ。わかりやすい番号でリビジョンを表すのでなく、単なる 40 文字の 16 進文字列を使うだって? さらに、リビジョンの並び順など存在しない。単に個々のローカルリビジョン内での並び順があるだけで、全体的な "並び順" は一直線ではなく DAG で表している。既に別の子チェンジセットを持つリビジョンに対して、新しい head をコミットして開発を続けることもできる。慣れ親しんだ中央集中型の VCS なら、そんなことをすれば警告が出ていたことだろう。

幸いにも、ツリーの並び順を可視化してくれるツールも存在する。また Mercurial では、チェンジセットのハッシュの (曖昧さのない) 短縮版も使えるし、**さらに**、ローカル限定ではあるが一連の番号でチェンジセットを識別することもできる。この一連の番号は単調増加する整数値で、クローンに投入されたチェンジセットの順番を表す。この順番はクローンごとに異なるので、リモートリポジトリに対する操作ではこの番号は使えない。

12.2 データ構造

DAG の概念がわかってきたところで、それが実際どのように Mercurial に格納されているのかを見ていこう。DAG モデルは Mercurial の内部動作の中核であり、いくつかの異なる DAG を使ってリポジトリをディスク上に格納している (コードのメモリ内での構造も同じだ)。このセクションではそれぞれの DAG について説明し、さらにそれらがどのように連携するかも説明する。

課題

実際のデータ構造に関する話題の前に、Mercurial の成長の歴史について簡単にまとめておく。Mercurial のそもそもの始まりは、Matt Mackall が 2005 年 4 月 20 日に Linux Kernel メーリングリストに投稿した一通のメールだった。その何日か前に、カーネルの開発で BitKeeper を使わないようにするという決断があったころの話である。Matt はそのメールで、いくつかの目標を示した。シンプルであること、スケーラブルであること、そして効率的であること。

[Mac06] において Matt が指摘したのは次のようなことである。今時の VCS は何百万ものファイルを含むツリーを扱う必要がある。チェンジセットの数もそれと同じくらいだろうし、何千ものユーザーが並行して新しいリビジョンを作ったりという作業を何十年ものスパンで行うことになるだろう。Mercurial が目指す目標を設定するにあたって、彼は現在の技術的な制約についてもまとめた。

- 速度: CPU
- 容量: ディスクやメモリ
- 帯域: メモリ、LAN、ディスクそして WAN
- ディスクのシーク率

ディスクのシーク率や WAN の帯域は今でも制約となっており、最適化を要する。この論文ではさらに、この手のシステムのパフォーマンスをファイルレベルで評価する際の一般的なシナリオや制約についての考察が続く。

- ストレージの圧縮: ファイルの履歴をディスク上に保存するのに最適の圧縮方法は何か? どんなアルゴリズムを使えば、CPU 時間をボトルネックにしない範囲で I/O のパフォーマンスを最大化できるか?
- 任意のリビジョンのファイルの取得: 多くのバージョン管理システムが採用している格納方法だと、過去のリビジョンのファイルを大量に読み込まないと新しいリビジョンを再現できない (差分を格納している)。我々としては、それを改善しつつ過去のリビジョンも高速に取得できるようにしたい。
- ファイルのリビジョンの追加: 新たなリビジョンの追加は頻発する。新しいリビジョンを追加するたびに古いリビジョンを上書きするようなことは避けたい。そんなことをすれば、リビジョンが増えるにつれて速度が低下するからである。
- ファイルの履歴の表示: ある特定のファイルを変更したすべてのチェンジセットの歴史を見直せるようにしたい。そうすれば、アノテーション (あるファイルの個々の行が、どのチェンジセットで変更されたものかを確認すること) を行えるようになる (これは CVS の時代には `blame` と呼ばれていたが、その言葉の否定的な意味を排除するために、後のシステムでは `annotate` という名前に変わったものもある)。

この論文ではさらに、同様のシナリオについてプロジェクトレベルでの考察を続ける。プロジェクトレベルでの基本的な操作といえば、あるリビジョンのチェックアウトや新たなリビジョンのコミット、そして作業ディレクトリとの差分の検出などである。差分の検出などは特に、ツリーが大きくなると速度が低下しがちだ (Mozilla や NetBeans プロジェクトを見るとよい。どちらも必要に迫られたバージョン管理に Mercurial を採用している)。

高速リビジョンストレージ: Revlog

Matt が Mercurial で用いた解決策が *revlog* (revision log を縮めたもの) だ。revlog 方式を使えば、各リビジョンのファイルの中身 (そして、前のバージョンからの差分) を効率的に管理できる。アクセス時間を効率化 (ディスクのシークを最適化) し、ストレージのスペースも効率的に使い、先に述べた一般的なシナリオをうまく扱えなければならない。そのために、revlog はディスク上ではふたつのファイルに分かれている。インデックスファイルとデータファイルだ。

6 バイト	ハンクオフセット
2 バイト	フラグ
4 バイト	ハンク長
4 バイト	非圧縮時の長さ
4 バイト	ベースリビジョン
4 バイト	リンクリビジョン
4 バイト	親リビジョン 1
4 バイト	親リビジョン 2
32 バイト	ハッシュ値

表 12.1: Mercurial のレコードフォーマット

インデックスは固定長のレコードで構成されており、その詳細は表 12.1 のとおりである。固定長のレコードを使う利点は、ローカルのリビジョン番号がわかればそのリビジョンにダイレクトアクセスできる (常に一定の時間でアクセスできる) ということだ。つまり、インデックスファイルの所定の位置 (インデックス長 × リビジョン) を読めば、データの場所がわかる。インデックスをデータから分離したことで、インデックスデータの読み込みが高速化した。インデックスを読む際にファイルのデータをすべてシークする必要がなくなる。

ハンクオフセットとハンク長で指定するのは、読む込むデータファイルの量である。これを読めば、そのリビジョンの圧縮データを取得できる。元データを取得するには、まず最初にベースリビジョンを読み込んでから当該リビジョンまでの差分を適用していけばよい。ベースリビジョンをどのタイミングで更新するかについてはちょっと考慮した。このタイミングを決める基準は、累積した差分のサイズとそのリビジョンの非圧縮時のサイズの比較である (データの圧縮には zlib を使い、ディスクスペースを節約している)。差分群のサイズをこの方式で一定に制限することで、特定のリビジョンを再構築するときの手間をできるだけ軽減している。そのおかげで、大量の差分を適用するはめになることを回避しているのだ。

リンクリビジョンは、そのリビジョンが依存する revlog を最高レベルまでたどるために利用する (あとでもう少し詳しく説明する)。そして親リビジョンには、ローカルのリビジョン番号 (整数値) が格納されている。これを使えば、関連する revlog のデータを見つけるのも容易になる。ハッシュに保存するのは、このチェンジセットを指す一意な識別子だ。SHA1 ハッシュに必要なのは 20 バイトだが、ここでは 32 バイトを確保している。これは、将来の拡張性を考慮したものである。

三種類の **revlog**

revlog が履歴データに関する汎用的な構造を提供しているので、その上位レイヤーとしてファイルツリーを表すデータモデルを作成できる。このデータモデルは次の三種類の revlog で構成されている。*changelog*、*manifests* そして *filelogs* だ。changelog には各リビジョンのメ

タデータが含まれ、さらに manifest revlog へのポインタ (つまり、manifest revlog 内のあるリビジョンのノード id) も含まれている。また、manifest ファイルにはファイル名の一覧が記録されており、さらに各ファイルのノード id も含まれている。このノード id は、各ファイルの filelog 内のリビジョンを指すものである。Mercurial のコード内では、changelog や manifest そして filelog を表すクラスがそれぞれ用意されている。これらはどれも汎用の revlog クラスを継承したものであり、個々の概念をきれいに層化している。

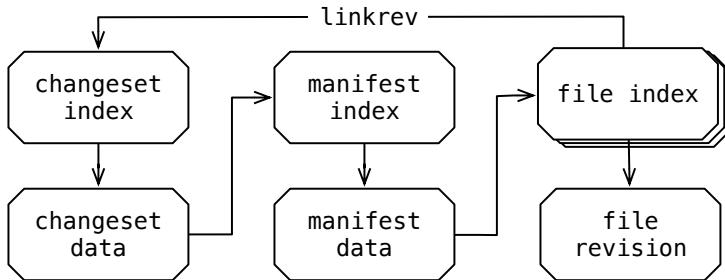



図 12.2: ログの構造

changelog のリビジョンは、このようになる。

```
0a773e3480fe58d62dcc67bd9f7380d6403e26fa
Dirkjan Ochtman <dirkjan@ochtman.nl>
1276097267 -7200
mercurial/discovery.py
discovery: fix description line
```

これこそが、revlog を層化したおかげで得られる価値である。changelog 層がこのようにシンプルな値のリストで表せるようになった。最初の行は manifest のハッシュである。それに続くのは作者名、そして日付と時刻 (Unix タイムスタンプとタイムゾーンオフセットをあわせた形式)、変更されたファイルの一覧、そして最後が内容を表すメッセージとなる。さらにもうひとつ隠し機能がある。実は任意のメタデータを changelog に含めることができるのだ。過去との互換性を維持するために、メタデータはタイムスタンプの後に追加することになる。

次は manifest だ。

```
.hgignore\x006d2dc16e96ab48b2fcca44f7e9f4b8c3289cb701
.hgsigs\x00de81f258b33189c609d299fd605e6c72182d7359
.hgtags\x00b174a4a4813ddd89c1d2f88878e05acc58263efa
CONTRIBUTORS\x007c8afb9501740a450c549b4b1f002c803c45193a
COPYING\x005ac863e17c7035f1d11828d848fb2ca450d89794
...
```

この manifest リビジョンは、changeset 0a773e が指している先である (Mercurial の UI は、曖昧にならない範囲で後半を省略できるようになっている)。これはツリーないのすべてのファイルを単純にならべた一覧であり、一行がひとつのファイルを表す。各行は、先頭にファイ

ル名があってその後に NULL バイトを置き、さらに 16 進エンコードしたノード id が続く。このノード id が各ファイルの filelog を指している。ツリーに含まれるディレクトリを特別に区別することはない。しかし、ファイルのパスにスラッシュが含まれていれば、それがディレクトリであろうと判断できる。manifest はその他の revlog と同様に差分を格納していたことを思い出そう。つまり、manifest は revlog 層から容易にアクセスできる構造でなければならない。任意のリビジョンに対して、そのリビジョンで変更があったファイルとその新しいハッシュだけを格納することになる。manifest は、通常は Mercurial の Python コードの中ではハッシュテーブル風のデータ構造で表される。ファイル名がキー、そしてノードが値である。

三種類めの revlog が、filelog だ。filelog は、Mercurial の内部的な `store` ディレクトリに格納されている。その名前は、追跡対象のファイルとほぼ同じである。ただし完全に同じではなく、ちょっとしたエンコードを行っている。主要な OS すべてできちんと動作させるようにするためである。たとえば、Windows や Mac OS X のように大文字小文字を区別しないファイルシステムも扱う必要があるし、Windows ではファイル名に使えない文字もある。また、ファイルシステムによってさまざまな文字エンコーディングが使われている。ご想像の通り、すべての OS で確実に動作させようとするとかなり大変なことになる。一方、filelog リビジョンの中身自体は、何の変哲もないものである。単にファイルの中身をそのまま保持しており、その先頭にちょっとしたオプションのメタデータが付属しているだけである (このメタデータを使って、ファイルのコピーやリネームなどの操作を追跡する)。

このデータモデルを使えば Mercurial リポジトリ内に格納されているどのデータにもアクセスできるようになる。しかし、このモデルがいつでも便利だとは限らない。実際の内部モデルは垂直指向 (すなわち、ファイルごとにひとつの filelog が存在する) である。だが、Mercurial の開発者がよくやりたくなる作業は、特定のリビジョンに関してすべてのファイルの詳細を扱うというものだった。そんな場合は、まず changelog から特定の changeset を探し、そのリビジョンに関連する manifest や filelog に簡単にアクセスできるようにしたい。そんな要望に応えるため、別のクラス群が用意された。これは revlog の上位層に位置するもので、まさに期待通りに役割を果たす。その名は `contexts` だ。

revlog の設定が分割されている利点のひとつが、順序づけをするときにあらわれる。順序づけをするには、まず最初に filelog を追加し、その後に manifest を続け、最後に changelog を追加する。その間リポジトリは常に一貫した状態となる。changelog を読み込むあらゆるプロセスは、その他の revlog へのすべてのポインタが有効であることを保証されていることになり、そのおかげでさまざまな問題に対応できるようになる。それでもなお、Mercurial には明示的なロックも存在する。このロックで、複数のプロセスが同時に並行して revlog に追記できないようにしている。

作業ディレクトリ

最後にもうひとつ、重要なデータ構造が残っている。その名は *dirstate* だ。dirstate とは、ある特定の時点で作業ディレクトリ内に何があるのかを表したものである。最も重要なこと

として、dirstate はどのリビジョンがチェックアウトされているのかを保持している。これが status や diff といったコマンドでのすべての比較の基準となり、さらに次にチェックインするコミットの親を決めることになる。dirstate は、merge コマンドを発行したときには二つの親を持つことになる。そして、一方の変更群を他方にマージしようと試みる。

status や diff はとても頻繁に使う操作なので (直近のチェンジセットから現時点でどれくらい進んでいるのかを確かめられる)、dirstate には Mercurial が最後に作業ディレクトリを走査したときの状態のキャッシュも残している。最終更新時刻とファイルサイズを追跡することで、ツリーの走査を高速化しているのだ。また、ファイルの状態も同時に追跡しなければならない。つまり、そのファイルが作業ディレクトリに追加されたのかあるいはそこから削除されたのか、変更が加わったのかといったことだ。これも作業ディレクトリの走査を高速化させ、コミット時にこれらの情報を取得しやすくしている。

12.3 バージョン管理の仕組み

ここまでで、Mercurial の内部的なデータモデルや低レベルでのコードの構造についてなじみが出てきたことだろう。さらにもう一歩進んでみよう。ここでは、これまでに説明した基盤の上で Mercurial がどのようにバージョン管理の概念を実装しているのかを考えていく。

ブランチ

ブランチの一般的な利用法は、開発のラインを別々に分割して進めて後で統合するというものだ。その理由として考えられるのは、たとえば誰かが新しい手法を試すときに開発のメインラインは常に出荷可能な状態にしておくこと (フィーチャブランチ)、あるいは旧バージョン用のバグフィックスを素早くリリースできるようにしておくこと (保守ブランチ) などである。どちらの手法もよく使われるものであり、今時のバージョン管理システムならすべてこれらの手法に対応している。暗黙のブランチは DAG ベースのバージョン管理では一般的なものだが、名前付きブランチ (ブランチ名をチェンジセットのメタデータに記録する方式) はそれほど一般的ではない。

当初は、Mercurial ではブランチの名前を明示することができなかった。ブランチの処理は、単に別々のクローンを作ってそれを個別に公開するというだけのことだった。これは効率的だし理解しやすく、特にフィーチャブランチに関しては非常に便利だった。というのも、ブランチを作るオーバーヘッドがほとんどなかったからである。しかし大規模なプロジェクトでは、クローンであってもかなり重たい作業となった。大半のファイルシステム上にリポジトリのハードリンクが格納されていてもなお、個別の作業ツリーを作るのには時間がかかるし大量のディスク容量を必要とする。

これらの弱点に対応するため、Mercurial には第二のブランチ方式が追加された。チェンジセットのメタデータにブランチ名を含める方式である。branch コマンドが追加され、現在の

作業ディレクトリにブランチ名を設定できるようになった。このブランチ名は、次のコミットのときに用いられる。通常の `update` コマンドを使ってある特定の名前のブランチにアップデートすることもできるし、ブランチ上でコミットしたチェンジセットは常にそのブランチに関連するコミットとなる。この手法は**名前付きブランチ** *(named branches)* と呼ばれる。しかし当初はブランチを作るだけでそれを閉じる (ブランチ一覧からそのブランチを見えなくする) ことはできなかった。ブランチを閉じられるようになったのは、それから数リリース後のことだった。ブランチを閉じる機能の実装は、チェンジセットのメタデータにフィールドを追加することで行った。追加フィールドに、このチェンジセットがブランチを閉じるという情報を記録したのだ。あるブランチが複数の head を持つ場合は、それらをすべて閉じてからでないとブランチ一覧からそのブランチを消すことはできない。

もちろん、何かを成し遂げる方法はたったひとつではない (there's more than one way to do it)。Git における名前付きブランチの実装は Mercurial とは異なり、参照を使っている。参照とは Git の歴史上で別のオブジェクト (通常はチェンジセット) を指す名前のことである。これは、Git のブランチがその場限りの短期的なものであることを意味する。つまり、いったん参照を削除してしまえば、そこにブランチが存在した形跡は一切なくなってしまう。これはちょうど、別々の Mercurial のクローンを作って、一方のクローンをもう一方にマージしたのと同じ状態だ。この方式だとローカルでのブランチの操作が非常に簡単かつ軽量になり、ブランチ一覧がごちゃごちゃすることも避けられる。

この方式でのブランチ管理はどんどん広まり、今や名前付きブランチ方式や Mercurial 風のクローンによるブランチよりもずっと幅広く使われている。その流れを受けて、Mercurial にも `bookmarks` 拡張が用意された。将来のバージョンでおそらく Mercurial に組み込まれることだろう。この拡張は、バージョン管理対象外のシンプルなファイルを使って参照を追跡する。Mercurial のデータ交換に使っている wire プロトコルを拡張してブックマークも通信できるようにし、ブックマークもプッシュできるようにした。

タグ

ちょっと見た限りだと、Mercurial におけるタグの実装は奇妙に感じることだろう。`tag` コマンドを使って最初にタグを追加するときに、`.hgtags` というファイルがリポジトリに追加され、それをコミットする。このファイルの各行には、チェンジセットのノード id とそのチェンジセットノードに対応するタグ名が記録される。このようにして、タグ情報ファイルはリポジトリ内の他のファイルとまったく同様の扱いとなるのである。

このようにした重要な理由が次の三つだ。まず最初に、タグは変更できなければならない。間違いは必ず発生するものだし、間違ったタグの修正や削除ができなければならない。次に、タグ自体もチェンジセットの履歴の一部でなければならない。そのタグがいつ誰によってどういう理由で作られたのか、あるいは変更されたのかといった情報を見られれば有用だ。最後に、過去のチェンジセットにさかのぼってタグを設定できなければならない。たとえば、

バージョン管理システムからエクスポートしたリリース用の成果物に対して大規模なテストをしてから初めてリリースするというプロジェクトも存在する。

これらの特性はすべて、`.hgtags` の設計から容易に得られる。人によっては作業ディレクトリ内に`.hgtags` ファイルが存在するのを見て困惑するかもしれない。しかし、そのおかげでタグ付けの仕組みと Mercurial のその他の部分との統合 (同じリポジトリの別のクローンとの同期など) が非常にシンプルに行えるのだ。もしタグがソースツリーとは別になっているのなら (Git などがそうだ)、そのタグの出自を調べたり重複したタグの衝突を解決したりするための仕組みが別途必要になる。後者の状況はめったにないかもしれないが、そもそもそんな状況が問題になりすらしない設計があるのならそっちのほうがよいだろう。

これらすべてをうまく機能させるために、Mercurial は`.hgtags` ファイルに新しい行を追加することしかしない。これは、別々のクローンでタグが並行して作られたときのマージにも役立つ。任意のタグに対して最新のノード id が優先され、空のノード id を追加すると (空のノード id は空の root リビジョンを表し、すべてのリポジトリに共通して存在する)、それはタグを削除するのと同じ意味になる。Mercurial はリポジトリ内のすべてのブランチのタグの関係についても考慮し、新しいタグのほうを優先する。

12.4 全体的な構造

Mercurial の大半は Python で書かれている。ごく一部で C が使われているところもあるが、これはアプリケーション全体としてのパフォーマンスを考慮したものである。そのごく一部を除いた部分では、Python で書くほうが適していると考えた。なぜなら、上位レベルの概念を表現するには Python のような動的言語のほうがずっと容易だったからである。コードの多くは特にパフォーマンスを最優先に考えたものではないが、そんなことはあまり気にしていない。それと引き替えに、大半の部分のコーディングが簡単になっているのだから。

Python のモジュールは、各モジュールがひとつのファイルに対応している。モジュールの中には必要に応じていくらでもコードを書くことができ、このモジュールがコードの構造をまとめる鍵となる。モジュールの中で型を使ったり他のモジュールの関数を呼び出したりするときには、他のモジュールを明示的にインポートすることがある。`__init__.py` モジュールを含むディレクトリはパッケージと呼ばれ、そこに含まれるすべてのモジュールやパッケージが Python のインポーターに公開される。

Mercurial がデフォルトでインストールする Python のパッケージは `mercurial` と `hgext` の二つである。`mercurial` パッケージには Mercurial を実行するために必要なコアコードが含まれており、もう一方の `hgext` にはさまざまな拡張機能が含まれている。コアと一緒に配布すると有用だろうと思われるものである。しかし、これらの拡張を使いたい場合は、設定ファイルを自分で編集して拡張を有効化させなければならない (詳細は後述する)。

念のために言うが、Mercurial はコマンドラインアプリケーションである。つまり、インターフェイスはシンプルなものだということである。ユーザーは、`hg` スクリプトにコマンド

を指定して呼び出すことになる。各種コマンド (`log` や `diff`、あるいは `commit` など) にはいくつかのオプションや引数を受け取るものもある。また、すべてのコマンドに共通するオプションもある。次に、このインターフェイスを通して起こることは次の三つに分類できる。

- hg は、ユーザーに何かを問いかけたり状態を示したりするメッセージを出力する。
- hg は、コマンドラインプロンプトでさらに入力を待ち受けることもできる。
- hg は、別のプログラム (コミットメッセージ用のエディタやコードの衝突を解消するためのマージを支援するプログラムなど) を起動することもある。

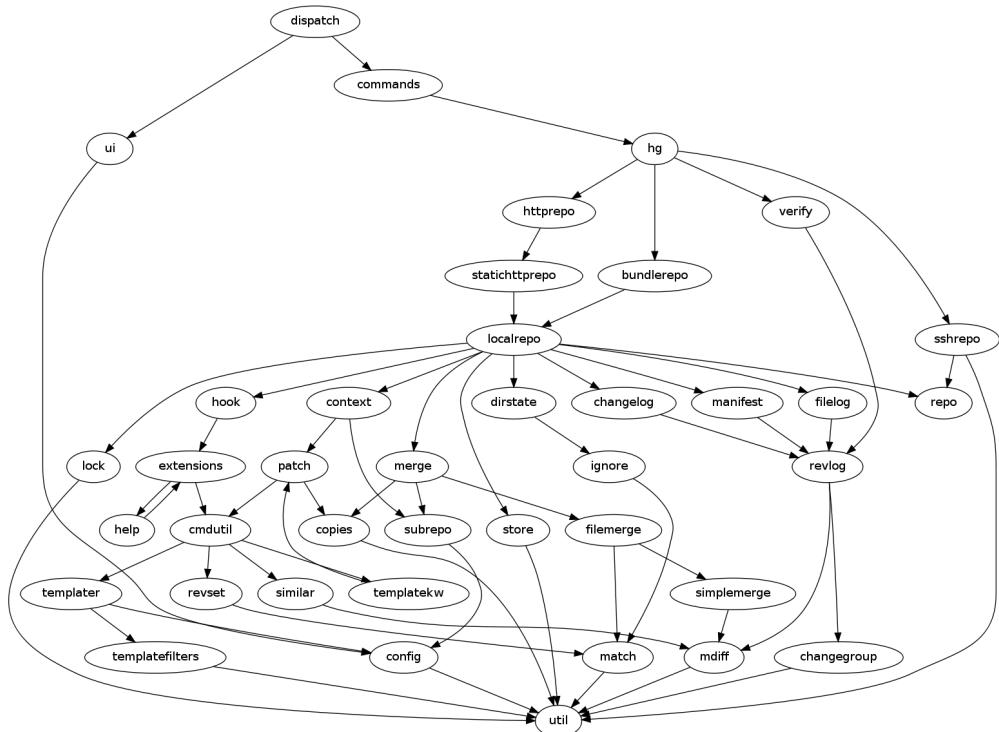



図 12.3: インポートグラフ

このプロセスのはじまりは、図 12.3 のインポートグラフから読み取れる。すべてのコマンドライン引数は、まず dispatch モジュールの関数に渡される。最初に行うのは `ui` オブジェクトのインスタンスを作ることである。`ui` クラスは、まず最初にいろいろな既知の場所 (ホームディレクトリなど) から設定ファイルを探し、設定オプションを `ui` オブジェクト内に保存する。設定ファイルには拡張機能へのパスが含まれていることもある。その場合は、この時点で拡張を読み込まねばならない。コマンドラインオプションで渡されたグローバルオプションもすべて、ここで `ui` オブジェクトに保存される。

これが終わると、次にリポジトリオブジェクトを作るかどうかを判断しなければならない。ほとんどのコマンドはローカルリポジトリ (`localrepo` モジュールの `localrepo` クラスで表

されるもの) が必要だが、なかにはリモートリポジトリ (HTTP や SSH、あるいはその他登録済みの形式でアクセスするもの) でも動作するコマンドもある。また、リポジトリを参照しなくても動作するコマンドもある。リポジトリを参照しないコマンドの例としては `init` があり、これは新しいリポジトリを初期化するときに使う。

すべてのコアコマンドは、`commands` モジュール内のひとつの関数で表される。そのため、何か特定のコマンドのソースコードを探すのも容易である。commands モジュールにはハッシュテーブルも含まれており、このハッシュテーブルではコマンド名とそれに対応する関数そしてそのオプションに関する説明を対応させている。こうすることで、一般的なオプション群 (たとえば、たいていのコマンドは `log` コマンドのオプションと同じようなオプションを使えるようになっている) を共有できるようになる。オプションに関する説明は、dispatch モジュールがそのコマンドのオプションをチェックするときに使う。そして、渡された値をそのコマンドの関数が期待する型に変換する。ほとんどすべての関数は、`ui` オブジェクトと操作対象の `repository` オブジェクトを受け取る。

12.5 拡張性

Mercurial をさらに強化できるように用意されたのが、拡張を書く機能である。Python は比較的身につけやすい言語であり、Mercurial の API はとてもうまく作られている (きちんとドキュメントも用意されている)。そんなこともあって「Mercurial の拡張を書きたいから Python を勉強しはじめました」という人も多い。

拡張機能の作成

拡張を有効にするには、Mercurial が起動時に読み込む設定ファイルの中に一行追加する必要がある。拡張を表すキーと、Python モジュールへのパスをそこに記述する。機能を追加するには次のようにいくつかの方法がある。

- 新しいコマンドの追加
- 既存のコマンドのラッピング
- 使用するリポジトリのラッピング
- Mercurial の関数のラッピング
- 新しいリポジトリ型の追加

新しいコマンドを追加するには、単に `cmdtable` という名前のハッシュテーブルを拡張モジュールに追加するだけである。このハッシュテーブルを拡張ローダーが読み込んで、コマンドテーブルに追加する。コマンドのディスパッチ時に、このコマンドテーブルを用いる。同じく、拡張内で `uisetup` および `reposetup` 関数を定義することもできる。これらはそれぞれ、UI やリポジトリのインスタンスを生成した後にディスパッチのコードから呼び出される。共通のふるまいが一つある。それは、`reposetup` 関数を使い、拡張が提供する repository

のサブクラスにリポジトリをラップすることだ。こうすることで、拡張側ですべての基本的なふるまいを変更できるようになる。たとえば、私がかつて書いたある拡張では、uisetup をフックして `ui.username` の内容を書き換えていた。環境から取得できる SSH 認証情報に基づいたユーザー名を設定していたのだ。

新しいリポジトリ型を追加するといった、さらに思い切った拡張を書くこともできる。たとえば `hgsubversion` プロジェクト (Mercurial 本体には組み込まれていない) は、Subversion のリポジトリを扱うためのリポジトリ型を登録する。これを使えば Subversion のリポジトリもクローンでき、まるで Mercurial のリポジトリであるかのように扱えるようになる。変更を Subversion のリポジトリにプッシュすることも可能だ。しかし両者のシステムの間にはインピーダンスミスマッチがあるため、かなりのエッジケースが存在する。一方、ユーザーインターフェイスに関しては完全に透過的になっている。

Mercurial を根本的に書き換えたい人には "モンキーパッチ" という手段がある。動的言語の世界ではおなじみの方法だ。拡張のコードは Mercurial と同じアドレス空間で実行され、また Python はリフレクション機能を持つ極めて柔軟な言語なので、Mercurial 内部で定義されている関数やクラスの書き換えだって容易に行える。見苦しいハックになってしまう可能性もあるが、非常に強力な仕組みでもある。たとえば、`hgext` にある `highlight` 拡張は、組み込みのウェブサーバーを書き換えてリポジトリブラウザにシンタックスハイライト機能を追加し、ファイルの中身を読みやすくしている。

Mercurial を拡張する方法は、もう一つ存在する。よりシンプルな方法である**エイリアス**だ。すべての設定ファイルでエイリアスを定義できる。エイリアスを使えば、既存のコマンドと設定済みのオプション群をあわせたものに新しい名前をつけることができる。これを使えば、既存のコマンドの短縮形を定義することもできる。最近のバージョンの Mercurial では、シェルのコマンドもエイリアスとして呼べるようになった。これを使えば、シェルスクリプトを使ってより複雑なコマンドを作ることもできる。

フック

各種バージョン管理システムは、これまでもずっとフック機構を提供してきた。この仕組みを使って VCS 上でのイベントとその他の世界とのやりとりをできるようにしていたのだ。よくある使い道としては、継続的インテグレーションシステム[1] に通知を送ったり、ウェブサーバー上の作業ディレクトリを更新して最新版を一般公開したりといったものがある。もちろん Mercurial にも同様に、フックを起動するサブシステムが組み込まれている。

実際のところ、フックにもまた二通りの仕組みが用意されている。一つは昔ながらの仕組みで他のバージョン管理システムにもよくあるもの、つまり、シェル内でスクリプトを実行するという仕組みだ。もう一方はもう少し興味深い仕組みである。というのも、ユーザー側で Python のフックを起動させる仕組みとして、Python のモジュール名とそのモジュールから呼び出す関数名を指定させているのだ。Mercurial と同じプロセス内で動くためにより高速

[1] 第 6 章を参照。

になるというだけでなく、この方式では `repo` オブジェクトや `ui` オブジェクトも渡せるので VCS ないでより複雑な操作を容易に行えるということになる。

Mercurial のフックは pre-command、post-command、controlling そして miscellaneous の四種類に分類できる。最初の二つは単純なもので、設定ファイルの hooks セクションに pre-*command* あるいは post-*command* というキーを設定してそこに任意のコマンドを定義するだけである。残りの二つについては、定義済みのイベントが用意されているのでそれを利用する。controlling フックがその他と異なる点は、何かが起こる直前にそのフックが実行され、場合によってはそのイベントの発生を中断させることもできるというところだ。よくある利用法は、中央サーバー上で何らかの方法でチェンジセットを検証するというものだ。Mercurial は分散型のシステムなので、コミット時にはこの手のチェックをすることができない。たとえば Python プロジェクトでは、フックを使ってコーディング規約のチェックを行っている。あるチェンジセットで追加しようとしているコードが所定のスタイルを満たしていない場合には、中央リポジトリがそれを却下するという仕組みだ。

それ以外のフックの使い道としては、プッシュログがある。これは Mozilla やその他多数の企業組織で使われている。プッシュログは個々のプッシュを記録し (ひとつのプッシュには複数のチェンジセットが含まれることもある)、さらにそのプッシュを誰がいつ行ったのかも記録する。これを、そのリポジトリの監査証跡とするのだ。

12.6 学んだこと

Matt が Mercurial の開発を始めるにあたって最初に決めたことのひとつが、Python で開発するということだった。Python は拡張性に優れており、非常にお手軽にコードを書ける。また、さまざまなプラットフォームでの互換性もきちんと考慮されているので、Mercurial を主要三大 OS で動くようにするのも比較的容易だった。その一方で、Python は他の (コンパイル型の) 言語に比べて実行速度が遅い。特にインタプリタの起動には時間がかかる。長期間実行しつづけるプロセスならこれは気にならないが、バージョン管理システムのように短期間に何度も起動するツールにとっては問題だ。

開発初期の方針として、いったんコミットした後でのチェンジセットの変更をしにくいようにした。あるリビジョンを変更するにはその id ハッシュの変更が必須になるので、いったんインターネット上に公開したチェンジセットの “取り消し” は面倒な作業となる。そこで Mercurial では、その作業が難しくなるようにしたのだ。しかし、まだ公開していないリビジョンを変更するぶんには特に問題はない。そこで、Mercurial がリリースされて間もなく、未公開のリビジョンの変更をしやすくするようコミュニティが動き出した。この問題を解決しようとする拡張も存在するが、利用方法を身につけるのが難しく、それまでふつうに Mercurial を使ってきたユーザーにとってはあまり直感的ではないものだった。

revlog はディスクのシーク処理を減らすのに役立ったし、changelog や manifest そして filelogs のレイヤー化アーキテクチャはうまく機能した。コミットは高速に行え、各リビジョンが利

用するディスク領域は比較的少なめになる。しかし、ファイルのリネームのような一部の作業は、各ファイルのリビジョンを別々に格納しているせいで効率的に行えない。最終的にはこの問題も修正されるだろうが、レイヤー構造に反するちょっとしたハックが必要になるだろう。同じく、ファイル単位の DAG を使って filelog ストレージを管理していることから、あまり大量のファイルを扱うのは現実的ではない。大量のファイルを管理するコードの一部がオーバーヘッドになってしまっている。

Mercurial が重視しているもう一つのことは、使い方を簡単に身につけられるようにすることだ。必須機能の大半を少数のコアコマンド群にまとめ、各コマンドで一貫性のあるオプションを用意している。その狙いは、特に他の VCS を使ったことがあるユーザーが Mercurial を段階的に学べるようにすることである。この思想の一環として、拡張機能で Mercurial をカスタマイズできるようにした。単に特定の使い道にあわせて拡張するだけではなく、それ以外にも使えるようにしたのだ。この狙いがあるため、Mercurial の開発者はその UI をできるだけ他の VCS(特に Subversion) と合わせようとしている。同様に、開発チームではドキュメントも重視している。Mercurial 自身に付属するドキュメントでは、他のトピックやコマンドへのクロスリファレンスも提供されている。また、エラーメッセージも有用なものになるよう心がけており、単に「操作が失敗しました」ではなく、何をすればいいのかというヒントを提供できるようにしている。

もう少し小さめの選択の中には、新しいユーザーにとっては少しびっくりするかもしれないものもある。たとえば、タグの処理を (先のセクションで述べたように) 作業ディレクトリ内の個別のファイルで扱うという方式を好まない人も多いだろう。しかしこの方式には好ましい特性もあるのだ (もちろん欠点もあるが)。また、他の VCS ではチェックアウトしたチェンジセットとその先祖だけをリモートホストに送るのがデフォルトだが、Mercurial ではリモートに存在しないすべてのチェンジセットを送信することを選んだ。どちらの方式にもそれなりの理由があり、あなたにとってどちらが最適かは、その開発スタイルに依存する。

他のソフトウェア開発プロジェクトと同様、Mercurial の開発でもさまざまなトレードオフが発生した。Mercurial は今までもよい選択をしてきたと思っているが、今になって考えると「もっと適切な選択肢があった」と思えるものもある。歴史的には、Mercurial は一般向けに使えるようになった第一世代の分散型バージョン管理システムである。個人的には、次の世代のシステムがどのようなものになるのかを楽しみにしている。

第13章

NoSQLを取り巻く世界

Adam Marcus

他の大半の章とは異なり、NoSQL は単体のツールを表す言葉ではない。時には補完しあったり時には競合したりするさまざまなツール群からなる生態系を指す用語である。NoSQL と名付けられたツール群は、SQL ベースのリレーショナルデータベースとは異なる方式でデータを格納する仕組みを提供する。NoSQL を理解するには、まずどんなツールが存在するのかを理解し、そしてそれぞれのツールがどのようにデータを格納するのかという設計を知る必要がある。

NoSQL をストレージシステムとして使おうと検討するときにまず理解すべきことは、NoSQL の中にもさまざまなシステムが存在するということである。NoSQL システムは、伝統的なリレーショナルデータベースシステムが持つ快適な機能の多くを廃止した。そして、これまでデータベース側に隠蔽されていた操作をアプリケーションの設計側に押し出したのだ。つまり、システムアーキテクトの立場で考えると、これらのシステムの仕組みをより深く知っておく必要があるということだ。

13.1 その名の由来は?

NoSQL の世界を語る前に、まずはその名前をきちんと定義してみよう。NoSQL システムとは、文字通りにとらえると SQL ではない問い合わせインターフェイスを持つシステムのことである。ただ、NoSQL コミュニティではもう少し包括的にとらえている。NoSQL システムとは今までのリレーショナルデータベースの代替となるものであり、開発者がシステムを設計するときに、SQL にとらわれずに (*Not Only* SQL) いろいろなインターフェイスを使えるようにするものだという考え方である。リレーショナルデータベースをその代替となる NoSQL で完全に置き換えることがあるかもしれないし、両者を組み合わせてアプリケーション開発時のさまざまな問題に対応していくこともあるかもしれない。

NoSQL の世界に飛び込む前に少し考えてみよう。SQL や関係モデルが適するのはどのような場面だろうか。そして NoSQL システムのほうがより適しているのはどんな場合だろうか。

SQL と関係モデル

SQL は、データを問い合わせるための宣言型の言語である。宣言型の言語とは、そのシステムに**何を**させたいのかをプログラマーが指定する言語のことである。そのシステムが**どのように**動くべきなのかを手続き的に定義するのではない。いくつか例を示そう。39 番の社員を探したり、レコード全体から社員名と電話番号だけを取り出したり、経理部門に属する社員のレコードだけに絞り込んだり、部署ごとの社員数を調べたり、社員テーブルのデータを管理職テーブルと連結させたりといった操作だ。

おおざっぱに言うと、SQL を使えば、データのディスク上での配置や使うインデックスそしてデータを処理するアルゴリズムを知らなくてもこれらの問いに答えられるということである。多くのリレーショナルデータベースのアーキテクチャ上で重要となるパーツが**クエリオプティマイザ**だ。これは、論理的に等価であるいろいろな問い合わせプランの中から最も効率的な問い合わせができるものを見つける。このオプティマイザは、たいていの場合はふつうのデータベースユーザーよりも賢い。しかし時には、必要な情報が不足していたりデータモデルがあまりにもシンプルすぎたりといった理由で最適な実行計画を生成できないこともある。

現在もっとも一般的に使われているデータベースがリレーショナルデータベースで、これは**関係データモデル**に従っている。このモデルは、現実世界のさまざまなエンティティをそれぞれ別のテーブルに格納する。たとえば、社員の情報は Employees テーブルに格納し、部署の情報は Departments テーブルに格納するといったものだ。テーブルの各行は、さまざな項目を保持している。たとえば社員の持つ情報としては社員 ID や給与、生年月日、そして姓名などである。これらの項目が、Employees テーブルの各カラムに格納される。

関係モデルは SQL と密接に関連している。フィルタのような単純な SQL 問い合わせは、あるフィールドが何らかの条件 (例: employeeid = 3、salary > $20000) にマッチするレコードをすべて取得する。層少し複雑な構造を使って、データベースに他の作業をさせることもできる。複数のテーブルからのデータの結合 (例: 社員番号 3 番の社員が所属する部署の部署名は?) などである。それ以外にも、集約 (例: 従業員の給与の平均額は?) などの操作もでき、この場合はテーブル全体のスキャンが発生する。

関係モデルでは高度に構造化されたエンティティを定義し、さらにそれらの間に厳格なリレーションシップも定義する。このようなモデルに対して SQL で問い合わせをすれば、カスタム開発なしで複雑なデータの取得もできるようになる。しかし、このように複雑なモデルや問い合わせにも制限はある。

- 複雑さは予測不可能性につながる。SQL は表現力がありすぎるので、個々のクエリのコスト、つまりその作業量に関するコストをきちんと考えないといけなくなる。問い合わせ言語をシンプルにするとアプリケーションのロジックが複雑になってしまうが、データストレージシステムを用意するのは簡単になる。単にシンプルなリクエストに応答できればそれで済むのだから。

- 問題をモデル化する方法は一つではない。関係データモデルは厳格なモデルであり、各テーブルに設定されたスキーマが個々の行のデータを規定する。あまり構造化されていないデータを格納する場合や行によって格納するカラムにばらつきがある場合などは、関係モデルだと制約が大きすぎることになる。アプリケーションの開発者の視点で考えても、関係モデルがあらゆる種類のデータを完璧にモデリングできるとは言えない。たとえば、アプリケーションのロジックの多くはオブジェクト指向の言語で書かれており、より高レベルの概念であるリストやキュー、セットなどを使っている。プログラマーの中には、永続層でこれらをモデリングしたいという人もいるだろう。
- データの量が増加して一つのサーバーでは保持しきれなくなると、データベース内のテーブルをパーティションに区切って複数のコンピューターで管理する必要がある。別のテーブルにあるデータを取得するためにネットワークをまたがる JOIN をするなどということは回避するには、テーブルを非正規化しなければならない。非正規化とは、さまざまなテーブルにあるデータの中で一度に使いたいものをすべて一か所にまとめて格納するということである。これにより、データベースはまるでキーで検索する方式のストレージシステムのようになるが、他にもっと適したデータモデルがないのかという思いは残る。

長年にわたって検討されてきた設計を独断で切り捨ててしまうのは、あまりよい考えではない。データをデータベースに格納するときには SQL と関係モデルを検討しよう。これは何十年にもわたる研究と開発のたまものであり、高度なモデリングができる。また、複雑な操作も容易に理解できることは請け合う。NoSQL が選択肢にあがるのは、何か特有の問題がある場合だ。たとえば大量のデータを扱う必要があったり作業量が膨大になったり、SQL とリレーショナルデータベースではうまく最適化できないようなデータモデリングを採用した場合などである。

NoSQL のはじまり

NoSQL ムーブメントの起源をたどれば、その大半は研究コミュニティの論文に行き着く。NoSQL システムの設計に関する決断には多くの論文が絡んでいるが、中でも特筆すべきなのが次の二つである。

Google の BigTable [CDG+06] は興味深いデータモデルを提示した。このモデルでは、複数列からなる履歴データを分類して格納する。データを複数のサーバーに分散させるために階層型のレンジベースパーティショニング方式を用い、データは厳密な整合性 (この概念については 13.5 節で定義する) のもとで更新される。

Amazon の Dynamo [DHJ+07] は、キー指向の分散型データストアを用いる。Dynamo のデータモデルはシンプルで、アプリケーション固有のデータの blob にキーをマッピングする。パーティショニング方式は障害からの回復機能を持つが、それを実現するためにデータの整合性はより緩やかな手法 (結果整合性) で管理している。

これら二つの概念についてこれから詳細を説明するが、その概念の多くは互いに組み合わせて使えるものだと理解しておくことが大切だ。たとえば、NoSQL システムのひとつである HBase[1] は、BigTable の設計に忠実な作りになっている。別の NoSQL システムである Voldemort[2] は、Dynamo の機能の多くを再現している。さらに別の NoSQL システムである Cassandra[3] の場合は、BigTable から引き継いだ機能 (データモデル) もあれば Dynamo から引き継いだ機能 (パーティショニングや整合性管理の方式) もある。

特徴と検討事項

NoSQL システムは、大掛かりな SQL 標準規格と決別して、ストレージの設計に関してシンプルながらも段階的なソリューションを提供する。その思想は、「データベースがデータを操作する方法を単純化すればするほど、アーキテクトは問い合わせのパフォーマンスを予測しやすくなる」というものだ。NoSQL システムの多くは、複雑な問い合わせロジックをアプリケーション側に任せている。その結果、データストア側では問い合わせのパフォーマンスを予測しやすくなる。問い合わせの種類が限られてくるからである。

NoSQL システムは、リレーショナルデータに単に宣言型の問い合わせ機能を追加するものとは一線を画する。トランザクション特性や整合性、永続性といった機能は、銀行などの組織のデータベースに求められる要件を満たすことを保証するものだ。**トランザクション**は、複数の込み入った操作をひとまとめにして「すべて成功」か「すべて失敗」かのいずれかになることを保証する。たとえば、ある口座から引き落とした資金を別の口座に入金するなどといった操作がトランザクションの対象となる。**整合性**は、ある値が更新されたときにそれ以降の問い合わせが更新後の値を見られることを保証する。**永続性**は、ある値が更新されたときにそれが安定したストレージ (ハードディスクドライブなど) に書き込まれ、データベースがクラッシュしても復旧可能であることを保証する。

NoSQL システムでは、これらの保証のうちのいくつかをより緩やかにした。金融関連のアプリケーションを除く多くのアプリケーションではそれでも受け入れ可能なレベルであり、保証を緩める引き替えにパフォーマンスを向上させている。このように保証を緩め、そしてデータモデルと問い合わせ言語を変更したことで、データベースをパーティショニングして複数サーバーに分散させることも簡単になった。データの量が増えて一つのマシンではさばききれなくなってもだいじょうぶである。

NoSQL システムは、まだ生まれたばかりの幼年期にある。本章で取り上げるシステムにおけるアーキテクチャ上の判断は、さまざまなユーザーの要求をとりまとめたものである。さまざまなオープンソースプロジェクトのアーキテクチャをまとめようとしたときに最も大変だったのは、どのシステムも変わりつつあるというところだった。個々のシステムの詳細は変わるものだということを頭に入れておこう。何かの NoSQL システムを使うことになった

[1] http://hbase.apache.org/
[2] http://project-voldemort.com/
[3] http://cassandra.apache.org/

ときに、本章はどう考えればいいかの指針にはなるだろう。しかし、その製品にどんな機能があるかを知るといった目的には使えない。

NoSQL システムについて考えるにあたって、検討すべき点を次にまとめる。

データモデルとクエリモデル: データの表現方法は? 行? それともオブジェクト? あるいはデータ構造とかドキュメントとか? データベースに対する問い合わせで複数のレコードを集約できる?

永続性: 値を変更したときに、それはすぐにストレージ上に反映される? ひとつのマシンがクラッシュした場合に備えて複数のマシンに格納する?

スケーラビリティ: データは単一のサーバー上に置く? データの読み書きをさばくために複数のディスクを扱う必要がある?

パーティショニング: スケーラビリティや可用性、永続性の観点から、データを複数のサーバーに置く必要がある? どのレコードがどのサーバーにあるのかということをどうやって知る?

整合性: 複数のサーバーにまたがってデータのパーティショニングや複製を行ったときに、レコードの変更に対して各サーバーはどのように協調する?

トランザクションの特性: 一連の操作を実行するときに、それをトランザクションとしてまとめることのできるデータベースもある。トランザクションは、実行中の他の操作との間の ACID (Atomicity: 原子性、Consistency: 整合性、Isolation: 独立性、Durability: 永続性) の一部あるいはすべてを保証する。これらの保証は一般的にパフォーマンスとのトレードオフとなるが、あなたが扱う業務ロジックはこれらの保証を要するものか?

単一サーバーでのパフォーマンス: データを安全にディスク上に格納したい場合に、ディスク上でのデータ構造として最適なものは? 読み込みが多い場合と書き込みが多い場合とではどうなる? ディスクへの書き込みがボトルネックになっている?

作業量の分析: ユーザーにとって使いやすいウェブアプリケーションを作るときには、作業量のチェックに細心の注意を払うことになる。多くの場合、欲しいのはデータセットサイズのレポートで、たとえば複数のユーザーの統計情報を集約したものだろう。あなたの利用法や使うツールは、その手の機能を必要としている?

これらすべての検討事項について取り上げる予定だが、後半の三点については (どれも同様に重要ではあるけれども) 本章ではあまり詳しく扱わない。

13.2 NoSQL のデータモデルおよびクエリモデル

データベースの**データモデル**とは、データをどのような論理構造で管理するかを示すものである。一方**クエリモデル**は、データの取得や更新をどのように行うのかを決定づける。一般的なデータモデルとしては、関係モデルやキー指向ストレージモデル、そして各種のグラフモデルなどがある。クエリ言語としては、SQL やキールックアップそして MapReduce などがおなじみだろう。NoSQL システムはさまざまなデータモデルとクエリモデルの組み合わせでできており、アーキテクチャ上の検討事項もそれぞれ異なる。

キーベースの **NoSQL** データモデル

NoSQL システムの多くは、関係モデルや SQL の機能性とは一線を画して、データセットの検索を単一フィールドによるものだけに制限している。たとえば、社員情報にはさまざまな項目があるにもかかわらず、ID による検索しかできないといった具合だ。その結果として、NoSQL システムにおける問い合わせの大半は、キーによる検索をベースにしたものとなる。プログラマーは、各データを識別するためのキーを選択する。そしてたいていの場合、できることといえばデータベース内でそのキーによる検索をしてアイテムを取得することくらいなのだ。

キールックアップ方式のシステムでは、複雑な結合操作や複数キーによる同一データの取得などを実現しようとすると、キーの名前にちょっとした工夫を要する。社員を検索する際に「社員 ID での検索」「ある部署に所属する全社員の検索」の二通りを行いたい場合は、二種類のキーを作成することになる。たとえば、キー `employee:30` は社員 ID が 30 の社員レコードを指し、キー `employee_departments:20` は部署番号 20 に属する全社員のリストを含むといった具合だ。結合操作はアプリケーションのロジック側に追い出される。部署番号 20 に属する全社員を取得するには、アプリケーション側でまず `employee_departments:20` を使って社員 ID のリストを取得し、そのリストをループさせて各 ID に対して `employee:ID` による検索を行う。

キールックアップモデルの利点は、データベースへの問い合わせのパターンが一貫したものになるというところである。データベースで行う作業はキールックアップだけであり、これは比較的一様で予測しやすいものである。アプリケーションのボトルネックを探すプロファイリングもシンプルになる。というのも、複雑な操作はアプリケーションのコード側に押し出されているからである。その反面、データモデルのロジックと業務ロジックが密結合してしまい、抽象化は崩れてしまう。

各キーに関連づけられたデータについて見ていこう。各種 NoSQL システムは、さまざまなソリューションを使っている。

キー・バリュー ストア

NoSQL のデータ格納方式の中で最もシンプルなのが**キー・バリュー**ストアだ。個々のキーを、任意のデータを含む値に対応させる。NoSQL システム自体はその値の中身については何も知らず、単にデータをアプリケーションに渡すだけとなる。先ほどの社員データベースの例で考えると、キー `employee:30` を blob に対応させることになる。その中身は JSON かもしれないし、Protocol Buffers[4] や Thrift[5] あるいは Avro[6] のようなバイナリフォーマットかもしれない。何らかの形式で、社員 ID が 30 の社員の情報をカプセル化したものとなる。

---

[4] `http://code.google.com/p/protobuf/`
[5] `http://thrift.apache.org/`
[6] `http://avro.apache.org/`

構造化されたフォーマットで複雑なデータを表してそれをキーに関連づけた場合は、取り出したデータの処理はアプリケーション側で行う必要がある。キー・バリュー形式のデータストアでは、キーを指定して取り出した価の中の特定の項目を取り出すような仕組みは用意されていない。キー・バリューストアの強みは、そのシンプルなクエリモデルにある。通常は `set`、`get` そして `delete` といったプリミティブで構成されている。一方、データベース内でのシンプルな絞り込み機能を追加する仕組みは持っていない。これは、扱うデータの中身が把握できないからである。Voldemort は Amazon の Dynamo をベースにしたシステムであり、分散型キー・バリューストアを提供する。BDB[7] は、キー・バリュー型のインターフェイスを持つ永続化ライブラリを提供する。

キー・データ構造 ストア

キー・データ構造 ストアは Redis[8] によって有名になった方式で、値として型を割り当てる。Redis では、値として使える型は integer、string、list、set そして sorted set である。`set/get/delete` に加えて型ごとに固有のコマンドも用意されている。たとえば整数型ならインクリメントやデクリメント、リストならプッシュやポップなどだ。さらに、クエリモデルにも機能が追加されている。リクエストの種類によってパフォーマンスが劇的に落ちるなどということはない。シンプルな、型ごとの機能を提供する一方で、集約や結合といった複数キーの操作はできない。このようにして、Redis は機能とパフォーマンスのバランスをとっている。

キー・ドキュメント ストア

キー・ドキュメント ストアには CouchDB[9]、MongoDB[10] そして Riak[11] などがある。これらは、構造化された情報を含むドキュメントをキーにマップする。これらのシステムは、ドキュメントを JSON 形式 (あるいはその類似形式) で格納する。リストや辞書も格納し、あるドキュメントの中に別のドキュメントを再帰的に埋め込むこともできる。

MongoDB はキー空間をコレクション内で分離しているので、たとえば Employees のキーと Department のキーが衝突することはない。CouchDB や Riak は、型の追跡を開発者側に任せている。ドキュメントの格納方法の自由さと複雑さは諸刃の剣である。アプリケーションの開発者はドキュメントのモデリングに関してかなりの自由を与えられているが、アプリケーション側での問い合わせのロジックは著しく複雑化する。

---

[7] `http://www.oracle.com/technetwork/database/berkeleydb/overview/index.html`
[8] `http://redis.io/`
[9] `http://couchdb.apache.org/`
[10] `http://www.mongodb.org/`
[11] `http://www.basho.com/products_riak_overview.php`

BigTable カラムファミリー ストア

HBase や Cassandra は、Google の BigTable が使っているモデルをベースにしたデータモデルを採用している。このモデルでは、キーがひとつの行を指し示す。その行に含まれるデータは、ひとつあるいは複数のカラムファミリー (CF) に格納されている。CF の中で、各行は複数の列を保持できる。列内の値にはタイムスタンプがついており、複数のバージョンの行・列マッピングを一つの CF に保持することができる。

概念的には、カラムファミリーを次のようにとらえることができる。つまり、ある形式 (行 ID、CF、列、タイムスタンプ) の複合キーを格納し、それをキーで並べ替えた複数の値にマップするというものだ。この設計が、多くの機能をキー空間に持たせるというデータモデリングにつながる。この方式は、タイムスタンプを持つヒストリカルなデータのモデリングに適している。もちろん、このモデルはスパース列に対応している。というのも、何も列を持たない行 ID が、必ずしも各列に NULL 値を持つ必要がないからである。その反面、NULL 値をほとんど (あるいはまったく) 持たない列でも各行に列 ID が必要となる。そのため、要する領域は多くなる。

どのプロジェクトのデータモデルもオリジナルの BigTable のモデルとはいろいろな面で異なるが、中でも特筆すべきなのが Cassandra における変更だろう。Cassandra は、各 CF の中にスーパーカラムという概念を導入した。スーパーカラムを使って、違ったレベルでのマッピングやモデリングそしてインデキシングを行えるようにしたのである。また、複数のカラムファミリーをひとまとめに格納してパフォーマンスを稼ぐ仕組みであるローカルグループという概念を廃止した。

グラフストレージ

NoSQL のデータ格納方式のひとつに、グラフストレージがある。データを作る方法はただ一つとは限らないし、リレーショナルモデルやキー指向のモデルがデータの格納や問い合わせに対して常に最適であるとは限らない。グラフは計算機科学では基本的なデータ構造であり、HyperGraphDB[12] や Neo4J[13] はグラフ構造のデータを格納する NoSQL ストレージシステムとしてよく知られている。グラフストレージは、これまで取り上げた他のストレージとはあらゆる点で異なる。データモデル、データの走査や問い合わせのパターン、ディスク上での物理的なデータ配置、複数マシンへの分散、クエリのトランザクション特性などがすべて異なるのだ。このように全く異なるものを公正に評価するにはページが足りない。しかし、これだけは意識しておこう。ある種のデータに関しては、グラフとして格納したほうがずっとうまく扱えるということを。

[12] http://www.hypergraphdb.org/index
[13] http://neo4j.org/

複雑な問い合わせ

NoSQL システムにおけるキーだけを使ったルックアップには、特筆すべき例外がある。MongoDB では任意の数のプロパティを使った索引付けができるようになっており、比較的高レベルの問い合わせ言語を使って取得したいデータを指定することもできる。BigTable ベースのシステムでは、スキャナーを使ったカラムファミリーの反復処理に対応しており、特定のアイテムを洗濯する際に、カラム上での絞り込みができる。CouchDB ではデータに対してさまざまなビューを作ることができ、テーブルに対して MapReduce タスクを実行させ、より複雑なルックアップや更新もできるようにしている。ほとんどのシステムには Hadoop あるいはその他の MapReduce フレームワークへのバインディングがあり、データセットに対して解析的な問い合わせを実行できる。

トランザクション

NoSQL システムは全般的に、**トランザクションの特性**よりもパフォーマンスを重視している。SQL ベースのシステムでは、複数の文の組み合わせ—主キーを指定して行を取得するといった単純な処理から、複数のテーブルを連結していくつかのフィールドの平均を算出するなどの複雑な処理まで—をひとつのトランザクションにまとめられるようになっている。

SQL データベースは、トランザクション間での ACID を保証している。複数の操作をひとまとめにして実行するトランザクションは原子性 (Atomic: ACID の A) がある。つまり、すべての操作が行われるか、なにも行われないかのいずれかになる。整合性 (Consistency: ACID の C) が保証するのは、そのトランザクションがデータベースの一貫性を保ち、状態を破壊しないということだ。独立性 (Isolation: ACID の I) とは、二つのトランザクションが同時に同じレコードを操作したとしてもお互い相手側に影響を及ぼさないということである。永続性 (Durability: ACID の D、次の節で詳述する) が保証するのは、トランザクションが確定したらそれが安全な場所に格納されるということだ。

ACID 準拠のトランザクションは、開発者に安心を与えてくれる。データの状態の確認が容易になるからだ。こんな場面を想像してみよう。複数のトランザクションが並行稼働しており、それぞれが複数のステップからなる処理をしている (たとえば、まず銀行口座の残高を確認してその次に$60 を引き落とし、最後に値を更新するなど)。ACID 準拠のデータベースはこれらの処理順序に関して何らかの制限がかかることが多いが、すべてのトランザクションで正しい結果が得られる。正確さを重視した結果、パフォーマンス特性に予期せぬ影響が出ることが多い。処理が遅いトランザクションが一つあるせいで、他のトランザクションもそれにあわせて処理を待つ羽目になるといったものだ。

大半の NoSQL システムは、ACID に完全に準拠することよりもパフォーマンスを向上させることを優先している。しかし、キーのレベルでは ACID を保証しており、同じキーに対する二つの操作があればそれは直列化され、キーと値のペアに深刻な被害が及ばないようにしている。多くのアプリケーションではこのレベルで十分で、データの正確性に目立った問題

が発生することもない。また、より規則性のある操作を素早く実行できるようになる。しかしこの方針のおかげで、アプリケーションの設計やデータの正確性に関して開発者側で考慮しなければならない点はより多くなる。

トランザクションを軽視する傾向に反する例外として特筆すべきなのが Redis だ。単一サーバー上で、Redis は `MULTI` コマンドを提供する。これは、原子性と整合性を保証した状態で複数の操作を組み合わせて実行するものだ。また `WATCH` コマンドは独立性を保証する。それ以外のシステムでは、より低レベルな *test-and-set*[14] 機能を提供しており、これで同様の独立性を保証している。

スキーマフリー ストレージ

多くの NoSQL システムに共通する特徴が、データベース内でスキーマを強要しないということである。ドキュメントやカラムファミリーを格納する方式であっても、各エンティティのプロパティが同じである必要はない。その利点は、構造化されたデータに関してサポートすべき要件が減るということ。そして、スキーマをオンザフライで修正するときにもパフォーマンスの劣化はあまり起こらないということである。そのぶん、アプリケーションの開発者側にはより多くの責務が課せられる。より身構えたプログラムが必要になるのだ。たとえば、社員レコードに `lastname` プロパティがなかったとして、それは修正すべきエラーなのだろうか。それとも、スキーマの更新がシステムを通じて伝搬している最中なのだろうか。*sloppy-schema* な NoSQL システムを使うプロジェクトの場合、数回のイテレーションを終えた後のデータやスキーマのバージョン管理は、アプリケーションレベルのコードで行うことになる。

13.3 データの永続性

理想を言えば、データに何か変更があったときにはそれをすぐに安全な場所に永続化させたいし、複数の場所にレプリカを作るなどしてデータのロスを防ぎたい。しかし、データの安全性を保証しようとするとパフォーマンスに影響が及ぶ。そこで、各種 NoSQL システムはそれとは異なる手法で**データの永続性**を保証しつつパフォーマンスを向上させている。データを失う場面にはさまざまなものがあるし、すべての NoSQL システムがこういった問題からあなたを守ってくれるというわけでもない。

いちばんシンプルかつありがちなシナリオは、サーバーの再起動や停電などだ。このような場合を想定してデータの永続性を確保するには、データをメモリからハードディスクに移すことになる。ハードディスクは、電源を落としてもデータを失わない。ハードディスク障害に対応するには、データを別のデバイスにコピーする。コピー先は、同一マシン上の別のハードディスク (RAID ミラー) だったりネットワーク上の別のマシンだったりする。しかし、データセンターも、ハリケーンなどの自然災害にあえば使えなくなる可能性がある。そこで、

[14][訳注]「更新前に確かめる」

ひとつのハリケーンで同時に被害にあうことのない程度に離れた場所にあるデータセンターにバックアップを取るという組織も存在する。データをハードディスクに書き込んで複数のサーバーやデータセンターにコピーするという作業は高くつく。そこで、さまざまなNoSQLシステムはデータの永続性の保証とパフォーマンスを天秤にかけてバランスを取っている。

単一サーバーの永続化

永続化の型として最もシンプルなのが**単一サーバーの永続化**で、サーバーを再起動したり電源を落としたりしても変更したデータが生き残ることを保証する。通常これは、変更したデータをディスクに書き込むことを意味する。そしてこの処理がボトルネックになることが多い。ディスク上のファイルにデータを書き込むようOSに指示を出したとしても、OSは書き込みをバッファリングしてすぐには書き込まないことがある。複数の書き込み操作を一括処理するためである。fsyncシステムコールを実行すれば、バッファにたまった更新をOSディスクに永続化させようと試みる。

一般的なハードディスクドライブの性能は、一秒あたり100から200のランダムアクセス(シーク)といったものであり、シーケンシャルライトもたかだか30-100 MB/sec程度である。どちらについても、メモリのほうが桁違いに高速である。単一サーバーでの永続化を保証する効率を上げるには、あなたのシステムによるランダムライトの回数を減らし、ハードディスクごとのシーケンシャルライトの回数を増やすようにすればよい。理想的には、一回のfsyncコールあたりの書き込み回数を最小化してシーケンシャルライトの回数を最大化したいものだ。そして、書き込みをfsyncさせるまではユーザーに対して書き込み成功を伝えないようにしておきたい。単一サーバーでの永続化を保証するときにパフォーマンスを改善するためのテクニックを、いくつか紹介する。

fsyncの頻度の制御

Memcached[15]は、ディスク上での永続化の保障を放棄する引き換えとして、極めて高速なインメモリでの操作を提供するシステムのひとつである。サーバーを再起動すると、memcached上のデータは消えてしまう。つまり、キャッシュとしてはよくできているが永続化には難があるデータストアとなる。

Redisの場合は、どのタイミングでfsyncをコールするかについていくつかのオプションを選べるようになっている。更新のたびにfsyncをコールするような設定もできる。これは、低速だが安全な選択肢となる。よりパフォーマンスを稼ぐために、N秒おきに書き込みをfsyncさせることも可能だ。この場合、最悪でN秒ぶんの操作をロストしてしまうことになるが、その程度なら受け入れられるという使い方もあるだろう。最後に、永続化を重視しないような場面(おおざっぱな統計情報の保守や、Redisをキャッシュとして使うといった場面)では、

[15]http://memcached.org/

fsync をまったくコールしないようにもできる。適当なタイミングで OS がデータをディスクに書き込むだろうが、それがいつ発生するかはまったく保証しないという選択肢だ。

ログ出力によるシーケンシャルライトの増加

NoSQL システムがディスクから高速にデータを取得するために、B+木などのデータ構造が使われている。こういったデータ構造のデータを更新するときには、ファイル内のランダムな場所を更新する。更新のたびに fsync しようとすると、一回の更新に対して複数のランダムライトが発生することになる。ランダムライトを減らすために、Cassandra や HBase、Redis、そして Riak といったシステムは、更新操作を *log* というファイルにシーケンシャルに書き込んでいる。システムで使っている他のデータ構造は言っての期間ごとに fsync するのに対して、ログだけは頻繁に fsync を行う。データベースがクラッシュしたときにはログを正式な状態として扱うことで、ランダムな更新をシーケンシャルライトにまとめている。

NoSQL システムの中には、MongoDB のようにその場でデータ構造に書き込みを行うものもあれば、さらにロギングを行うものもある。Cassandra や HBase が使っているテクニックは BigTable を参考にしたもので、ログとルックアップデータ構造をひとつの *log-structured merge tree* にまとめている。Riak は、同党の機能を *log-structured hash table* で提供する。CouchDB は伝統的な B+木に手を加えたものを使っており、すべての変更を物理ストレージ上の構造に追記する。これらのテクニックによって書き込みのスループットは向上するが、定期的にログの最適化をしないとログのサイズがどんどん膨れ上がってしまう。

書き込みのグルーピングによるスループットの向上

Cassandra は、複数の更新を並行してまとめ、一回の fsync コールの間に実行する。このような設計は**グループコミット**と呼ばれており、更新あたりの待ち時間が長くなってしまう。ユーザーによる更新が受け付けられたかどうかを知るには、並行するいくつかの更新が完了するまで待たなければならない。待ち時間が増える一方で、これはスループットの向上につながる。複数のログ追記処理が一回の fsync で処理されるからだ。本章の執筆時点では、HBase の更新は Hadoop Distributed File System (HDFS)[16] が提供するストレージに永続化される。最近適用されたパッチで、fsync やグループコミットを尊重する追記もサポートするようになった。

複数サーバーの永続化

ハードディスクドライブだってマシンだって、壊れてしまって復旧不能になることがある。重要なデータは別のマシンにもコピーしておくことが必須である。多くの NoSQL システムには、複数のサーバーを使ってデータを永続化する仕組みが存在する。

[16] http://hadoop.apache.org/hdfs/

Redis は、伝統的なマスター/スレーブ型の手法でデータを複製する。マスターに対してなされたすべての操作がログ風の仕組みでスレーブ機に送られ、スレーブ機の上で同じ操作を再現させる。マスター上で障害が発生したら、マスターから受け取ったオペレーションログの状態に基づいてスレーブがデータを提供することになる。この構成では、何らかのデータロスが発生する可能性がある。マスターへの更新の結果をユーザーに返す前に、スレーブ上へのログの永続化が完了したかどうかを確認しないからである。CouchDB は、同様の形式で双方向のレプリケーションを行う。ドキュメントに対する変更を他のマシンに複製するよう、サーバーを設定するのだ。

MongoDB にはレプリカセットという仕組みがあり、何台かのサーバーで各ドキュメントの格納にかかわる。MongoDB のオプションで、すべてのレプリカが更新を受け取ったことを保証させることもできる。一方、最新のデータがすべてのレプリカに行きわたるのを確認せずに処理を進めることもできる。その他多くの分散型 NoSQL ストレージシステムは、データのマルチサーバーレプリケーションに対応している。HDFS 上に構築される HBase は、複数サーバーの永続化を HDFS 経由で実現する。すべての書き込みは、ふたつ以上の HDFS ノードに複製されるまでユーザーに制御を返さない。これによって、複数サーバーでの永続化を保証している。

Riak や Cassandra そして Voldemort では、より細やかにレプリケーションを設定できる。それぞれ微妙な違いはあるが、これらのシステムでは `N` と `W` のふたつの値を設定できる。`N` は最終的にデータのコピーを保持することになるマシンの台数、そして `W` は `W<N` を満たす数で、少なくともこれだけの台数のマシンにデータが書き込まれた時点でユーザーに制御を戻す。

データセンター全体のサービスが停止してしまう事態に対応するには、複数のデータセンターにまたがるマルチサーバーのレプリケーションが必要になる。Cassandra や HBase そして Voldemort には *rack-aware* な設定があり、さまざまなマシンがどのラック (あるいはどのデータセンター) に配置されているのかを指定することができる。一般に、リモートサーバーでの処理が完了するまでユーザーのリクエストをブロックすると、待ち時間が長くなってしまう。そこで、WAN による別のデータセンターへのバックアップのときは、処理の完了を確認せずに更新処理を終える。

13.4　パフォーマンス向上のためのスケーリング

エラー処理について語る前に、もっと楽観的な状況を考えてみよう。やった!大成功!という場面だ。あなたが構築したシステムがうまく動き出すと、データストアはそのコンポーネントのひとつとなり、それなりの負荷にさらされることになる。お手軽だがあまり美しくない解決策は、既存の機器を**スケールアップ**することだ。RAM とディスクをさらに調達して、ひとつのマシンでさばける量を増やせばよい。あなたのシステムがさらに成功を収めると、ハードウェアをよいものにして高価なメモリをどんどん投入することにも限界が出てくる。ここまでくると、データをレプリケートして複数マシンで負荷分散をさせるしか方法がなく

なる。これは**スケールアウト**とよばれる手法で、システムの**水平スケーラビリティ**の指標となる。

水平スケーラビリティの理想的な目標は**リニアなスケーラビリティ**、つまり、ストレージシステムのマシン数を二倍にすればそのシステムのクエリ処理性能も二倍になるというものだ。これを実現するためのポイントは、データを複数マシンにどのように振り分けるかということになる。シャーディングという手法は、読み込みと書き込みを複数のマシンに分散させてストレージシステムをスケールアウトさせるものである。シャーディングは多くのシステムで設計の基本となっている。具体的には Cassandra や HBase、Voldemort、Riak、そして最近では MongoDB や Redis もそうだ。中には、CouchDB のように単一サーバーでのパフォーマンスを重視してシステムではシャーディング機能を提供しないというプロジェクトもある。しかし、セカンダリプロジェクトがコーディネーターとなって、複数のマシンにそれぞれ独立にインストールした環境に処理を振り分けることもできる。

ここで、いくつかの用語についてまとめておこう。ここでは、**シャーディング**と**パーティショニング**を同じ意味で使う。また、**マシン**や**サーバー**そして**ノード**は、分割されたデータを格納する物理的な計算機を指すものとする。最後に、**クラスタ**あるいは**リング**という用語は、ストレージシステムを構成するマシン群を指すものとする。

シャーディングするということは、どのマシンもそれ単体ではデータセット上のすべての書き込みを処理する必要がなくなるということだ。しかしそれと同時に、どのマシンもそれ単体ではデータセットへのすべての問い合わせには対応しきれないということでもある。多くの NoSQL システムではキー指向のデータモデルやクエリモデルを採用しているので、いずれにせよデータセット全体にまたがるような問い合わせはほとんどない。これらのシステムではデータにアクセスする主要な方法がキーに基づいているので、シャーディングもまたキーに基づいて行うのがよい。キーに対する何らかの関数が、そのキー・バリューペアをどのマシンに格納するのかを決定する。キーとマシンのマッピング方法を二通り紹介しよう。ハッシュパーティショニングとレンジパーティショニングだ。

必要になるまでシャーディングを避ける

シャーディングはシステムを複雑化させるものであり、可能な限り避けるべきである。ここでは、シャーディングを使わずにスケールさせる方法を二通り取り上げる。リードレプリカとキャッシュだ。

リードレプリカ

ストレージシステムの多くは、読み込みリクエストのほうが書き込みリクエストより多くなる。そんな場合のシンプルな解決策は、データのコピーを複数のマシン上に置くことだ。書き込みリクエストはすべてマスターノードに任せる。読み込みリクエストはデータのレプ

リカを持つマシンにまわす。ただしこのレプリカは、書き込みサーバー上のデータよりも若干古いものになることが多い。

もし既にマスター・スレーブ公正で複数サーバーでのデータの永続化を実現している (Redis や CouchDB そして MongoDB ではそれが一般的) のなら、読み込み用のスレーブが書き込み用のマスターの負荷を多少軽減させることができる。ある種のクエリ、たとえばデータセットのサマリーの集計などは、コストのかかる処理である一方で必ずしも最新のデータを必要とはしないことがある。そんなクエリは、スレーブのレプリカに対して実行すればよい。一般に、データが最新である必要性が少なければ少ないほど、その処理を読み込み用スレーブに任せてクエリのパフォーマンスを稼ぎやすくなる。

キャッシュ

システム上でよく使われるコンテンツをキャッシュすると、たいていの場合は驚くほどうまく機能する。Memcached は、複数サーバー上にメモリブロックを確保してデータストアのデータをキャッシュする。Memcached クライアントは、水平スケーラビリティの技を使って別のサーバー上にある Memcached に負荷を分散する。キャッシュプールにメモリを追加するには、単に Memcached が動くホストを追加するだけでよい。

Memcached はキャッシュ用に設計されているので、処理をスケールさせるための永続化ソリューションのアーキテクチャはそれほど複雑ではない。複雑なソリューションの前に、キャッシュでスケーラビリティの問題を解決できないかどうかを検討してみよう。キャッシュ処理は単なるその場しのぎのバンドエイドではない。Facebook では Memcached で何と数十テラバイトものメモリを確保しているのだってさ!

リードレプリカやキャッシュを使えば、読み込み処理をスケールアップさせることができる。しかし、書き込みやデータ更新の頻度が上がり始めたら、最新状態を保持するマスターサーバーへの負荷が増加することになる。このセクションの後半では、書き込み処理を複数サーバーにシャーディングする方法を扱う。

コーディネーターによるシャーディング

CouchDB プロジェクトは、単一サーバー上での挙動を重視している。Lounge と BigCouch のふたつのプロジェクトは外部のプロキシを通じて CouchDB への負荷をシャーディングし、単独の CouchDB インスタンスのフロントエンドとして機能する。この構成では、個々の CouchDB がお互いを意識することがない。コーディネーターが、リクエストされたドキュメントのキーに応じて個々の CoucdDB インスタンスにリクエストを分散させる。

Twitter は、シャーディングやレプリケーションの概念をまとめた Gizzard[17] というフレームワークを構築した。Gizzard は、任意の型のスタンドアロンデータストア (SQL システムあるいは NoSQL システムのラッパーを作れる) を受け取り、キーのレンジでパーティショニン

[17] http://github.com/twitter/gizzard

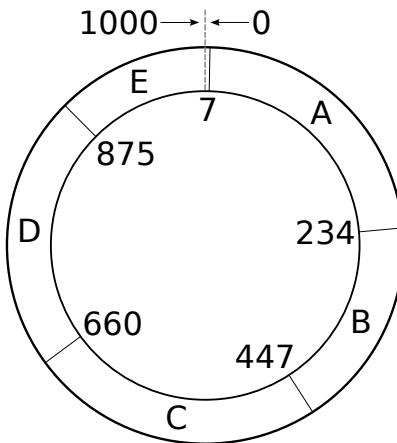



図 13.1: 分散ハッシュテーブルリング

グして任意の深さのツリーとしてまとめる。耐障害性を高めるため、Gizzard は同じキーレンジのデータを複数の物理マシンにレプリケートするようにも設定できる。

コンシステントハッシュリング

よくできたハッシュ関数は、キーのセットを統一された形式で分散させる。これは、キー・バリューのペアを複数のサーバーに分散させるための便利なツールとして使える。学術論文上で**コンシステントハッシュ**と呼ばれる技術は広範囲にわたる。このテクニックをデータストアに応用したはじめての例が**分散ハッシュテーブル** (*DHT: Distributed Hash Tables*) というシステムだ。Amazon の Dynamo の動作原理を参考にした NoSQL システムではこの分散テクニックを採用しており、Cassandra や Voldemort そして Riak などにそれが見られる。

ハッシュリングの実例

コンシステントハッシュリングは、次のように動作する。まず、ハッシュ関数 `H` があるとしよう。この関数は、キーを均等に分布する大きな整数値にマップする。比較的大きな整数値 L をとって範囲 [1, L] の数のリングを作れば、H(key) mod L をそのリングに含めることができる。これで、各キーが [1, L] の範囲に収まることになる。サーバーのコンシステントハッシュリングは、各サーバーの固有な識別子 (その IP アドレスなど) に H を適用したものを使って構成される。この動作原理を理解しやすくするために、5 台のサーバー (`A-E`) からなるハッシュリングの例を図 13.1 に示す。

ここでは、`L = 1000` とした。`H(A) mod L = 7`、`H(B) mod L = 234`、`H(C) mod L = 447`、`H(D) mod L = 660`、そして `H(E) mod L = 875` であるとしよう。これで、キーがどのサーバーに配置されるかがわかるようになった。リング内で、あるサーバーとその次のサーバーの間

にキーが収まる場合に、そのサーバーにキーを格納することになる。たとえば、A が受け持つキーはそのハッシュが [7, 233] の範囲になるものであり、E が受け持つキーはそのハッシュが [875, 6] の範囲 (これは、値が 1000 の部分をまたがっている) になるものである。つまり、もし H('employee30') mod L = 899 になるのならこのデータはサーバー E に格納されるし、H('employee31') mod L = 234 になるとしたらこのデータはサーバー B に格納される。

データのレプリケーション

複数サーバーでの永続化のためにレプリケーションをするには、あるサーバーの担当範囲に割り当てられたキーと値のペアをリング内でのその次のサーバーに渡せばよい。たとえば、三重のレプリケーションを行うには、範囲 [7,233] にマップされたキーをサーバー A、B、そして C に格納する。仮に A がダウンしたら、隣にある B と C がその範囲を受け持つ。設計によっては、E にレプリケートして A の担当分を一時的に受け持つこともある。この場合は担当範囲を拡張して A の範囲も含めるようにする。

よりよい振り分け

ハッシュはキー空間を均等分布させるという点では統計的に有効であるが、均等に分布させるには通常はある程度多くのサーバーを必要とする。残念ながら、たいていは少数のサーバーからスタートすることが多い。そしてその段階では、ハッシュ関数がうまくキーを分散させてはくれない。先ほどの例で見ると、A の担当範囲の長さは 227 であるのに対して E の担当範囲は 132 しかない。こんな状態だと、サーバーによって負荷が違うということになるだろう。また、どれかひとつのサーバーがダウンしたときに代わりを受け持つのも難しくなる。隣のサーバーが、ダウンしたサーバーの担当範囲全体を制御しなければならなくなるからである。

担当するキーの範囲にばらつきが出てしまう問題を解決するために、Riak を含む多くの DHT は、物理マシン単位でいくつか '仮想' ノードを作成する。たとえば、4 つの仮想ノードを作ったサーバー A が、サーバー A_1、A_2、A_3、そして A_4 として動作するといった具合だ。各仮想ノードには異なるハッシュ値が割り当てられ、キー空間の各部分によりうまく分散させられるようにする。Voldemort も同様の手法を採用しており、パーティションの数を手動で設定できるようになっている。通常はサーバーの数よりパーティションの数を多くするので、結果的に各サーバーが複数のパーティションを受け持つことになる。

Cassandra は、各サーバーで複数の小さなパーティションを担当するということはしない。そのため、時にはキーの範囲の分散が均一にならないこともある。ロードバランス用として、Cassandra には非同期プロセスが用意されている。このプロセスは、これまでの負荷の履歴に基づいてリング上でのサーバーの配置を調整する。

レンジパーティショニング

レンジパーティショニング方式によるシャーディングでは、システム内の何台かのマシンが「どのサーバーがどのキー範囲を受け持つか」というメタ情報を保持する。このメタ情報への問い合わせによって、キー範囲の検索と適切なサーバーへの振り分けを行う。コンシステントハッシュリングの手法と同様、レンジパーティショニングもキー空間をいくつかの範囲に分割する。そして各範囲をひとつのマシンが管理し、場合によっては他のマシンにレプリケートしたりする。コンシステントハッシュ方式と違うところは、キーのソート順で隣同士になるキーがほぼ同じパーティションに収まるという点だ。これにより、ルーティング用のメタデータのサイズを軽減できる。範囲を表すには単に [開始位置, 終了位置] の印があればよいだけだからである。

キー範囲とサーバーのマッピング情報を更新し続ける際に、レンジパーティショニング方式では高負荷なサーバーの負荷分散をよりきめ細やかに制御できるようになる。特定のキー範囲が他の範囲に比べてトラフィックが多くなるようなら、ロードマネージャーはそのサーバーが担当する範囲を狭めることもできるし、そのサーバーが担当するシャードの数を減らすこともできる。動的に負荷を調整できるという新たな自由を得るために使ったのは、シャードを監視したりルーティングしたりするための追加コンポーネントだ。

BigTable の手法

Google による BigTable に関する論文には、階層化レンジパーティショニングでデータをタブレットにシャーディングする手法が解説されている。一つのタブレットが、一定範囲の行のキーとカラムファミリー内の値を保持する。タブレットは必要なログをすべて保持し、自分が担当する範囲のキーに関する問い合わせに答えるためのデータ構造もすべて保持する。タブレットサーバーは、各タブレットにかかる負荷に応じて複数のタブレットを担当する。

各タブレットは、100 から 200MB のサイズを保つ。タブレットのサイズが変われば、隣り合うキー範囲の二つの小さなタブレットを一つにまとめたり、あるいは大きなタブレットを二つに分割したりする。マスターサーバーが、タブレットのサイズや負荷そしてタブレットサーバーの稼働状態を解析する。そして、マスターサーバーは、どのタブレットサーバーがどのタブレットを担当するのかを常時調節する。

マスターサーバーは、タブレットの割り当てをメタデータテーブルで管理する。このメタデータはかなり大きくなる可能性があるので、メタデータテーブル自体も複数のタブレットにシャーディングしてキー範囲とタブレットを関連づける。タブレットサーバーが、この範囲の管理を行う。その結果、クライアントは三階層の走査を経てキーの保存先のタブレットサーバーを知ることになる。その様子を図 13.2 に示す。

実際の例を見てみよう。クライアントがキー `900` を検索しようとすると、サーバー `A` に問い合わせを行う。このサーバーには、レベル 0 のメタデータ用のタブレットが格納されている。このタブレットを見れば、レベル 1 のメタデータあがサーバー 6 上のタブレットにあること

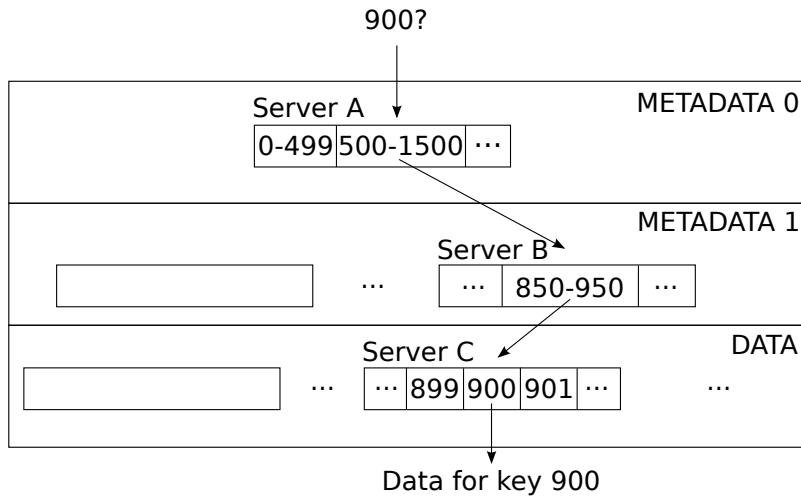



図 13.2: BigTable ベースのレンジパーティショニング

がわかる。このタブレットには 500-1500 の範囲のキーが含まれる。クライアントはサーバー B に対してこのキーでリクエストを送る。そして、キー 850-950 を含むタブレットがサーバー C 上にあるという応答を得る。最終的に、クライアントはそのキーについてのリクエストをサーバー C に送り、問い合わせ結果の行を取得する。レベル 0 とレベル 1 のメタデータタブレットはクライアント側でキャッシュしてもかまわない。そうすれば、タブレットサーバーに対する同じような問い合わせの繰り返しを回避できる。BigTable の論文では、この三段階の階層によって 2^{61} バイトのストレージを 128MB のタブレットで対応できるとしている。

障害の処理

BigTable の設計では、マスターが単一障害点になる。しかし、タブレットサーバーへのリクエストに影響が及ばないように一時的にダウンさせることも可能だ。タブレットへのリクエストを処理するタブレットサーバーがダウンすれば、マスターがそれを認識してそのタブレットの担当を切り替えるまではリクエストが一時的に失敗するようになる。

マシンの障害を認識して対応するための方法として、BigTable の論文では Chubby を使っている。これは分散ロックシステムで、サーバーの死活監視を行うものである。ZooKeeper[18] は Chubby のオープンソース実装で、Hadoop ベースのプロジェクトの中には、これを使ってセカンダリマスターサーバーやタブレットサーバーの再配置を行うものもある。

[18] http://hadoop.apache.org/zookeeper/

レンジパーティショニングベースの NoSQL プロジェクト

HBase は、BigTable の階層方式を使ってレンジパーティショニングを行う。ベースとなるタブレットのデータは Hadoop の分散ファイルシステム (HDFS) に格納する。HDFS がデータのレプリケーションやレプリカ間の整合性を管理する。そして、タブレットサーバー側ではリクエストの処理やストレージの更新、タブレットの分割や統合を管理する。

MongoDB も、BigTable と同様の方法でレンジパーティショニングを処理する。いくつかの設定ノードがルーティングテーブルを管理して、このルーティングテーブルを使ってどのストレージノードがどの範囲のキーを担当するかを決定する。設定ノード間の同期は**二相コミット**というプロトコルで行い、BigTable のマスターが受け持つ範囲指定機能と Chubby が受け持つ高可用性のための構成管理機能の両方を提供する。ステートレスで動くルーティングプロセスとは別に、直近のルーティング設定を追跡し続けることでキーへのリクエストを適切なストレージノードに振り分ける。ストレージノードはレプリカセット内に置かれ、レプリカセットでレプリケーションを行う。

Cassandra では順序を保持したパーティショニング機能を提供しており、データに対するレンジスキャンを高速に行いたい場合に利用できる。Cassandra のノードもコンシステントハッシュを使ってリング内に配置される。しかし、キー・バリューのペアをハッシュして割り当て先のサーバーを決めるのではなく、キーそのものを単純にサーバーにマップする。そうすることで、キーが必然的にフィットする範囲を制御できるようになる。たとえばキー 20 と 21 は、図 13.1 のコンシステントハッシュリング上でどちらもサーバー A にマップされる。ハッシュしてリング内でランダムに分散させるわけではない。

Twitter の Gizzard フレームワークは、分割され、レプリケートされたデータをさまざまなバックエンドにまたがって管理するもので、レンジパーティショニングを使ってデータをシャーディングする。ルーティングサーバーは任意のレベルの階層構造を構成し、キー範囲をその階層下のサーバーに割り当てる。各サーバーは自分に割り当てられたキー範囲のデータを格納したり、別の層のルーティングサーバーに処理を振ったりする。このモデルにおけるレプリケーションは、あるキー範囲の更新を複数のマシンに送信することで実現する。Gizzard のルーティングノードは、書き込みに失敗したときの対応方法が他の NoSQL システムとは異なる。Gizzard を使ったシステムでは、すべての更新が冪等 (何度実行しても結果が変わらない) でなければならない。ストレージノードがダウンしているときは、ルーティングノードがその処理をキャッシュし、更新が確認できるまで何度もその処理を送信し続ける。

どのパーティショニング方式を採用するか

ハッシュ方式とレンジ方式、シャーディングの手法としてどちらが適切なのかって? それは状況による。明らかにレンジパーティショニングを選ぶべきなのは、たとえばキーによる検索よりも範囲指定による検索が多発するような場面だ。キーの順に値を読み込むことになるので、そうしておけばネットワーク上のあちこちのノードを飛び回る必要がなくなる。ノー

ドを移るときのネットワークのオーバーヘッドは馬鹿にできない。しかし、範囲指定の検索が不要な場合には、どちらの方式を選べばよいのだろう?

ハッシュパーティショニングを使えば、データを複数のノードに適切に分散させることができる。そして、データの非対称性も仮想ノードで軽減できる。ハッシュパーティショニング方式では、ルーティングもシンプルになる。ほとんどの場合は、クライアント側でハッシュ関数を実行すれば問い合わせ先のサーバーを確定できる。バランスを調整するための仕組みを導入している場合は、あるキーに対して適切なノードを見つけるのは少し難しくなる。

レンジパーティショニングは、ノードのルーティングや構成を管理するための事前コストが必要となる。この処理の負荷は高くなりがちで、耐障害性をきちんと考慮していないと障害の主因になるだろう。しかし、うまくやれば、レンジパーティショニングで分割したデータは小さいチャンクで負荷分散できるようになり、負荷が高くなったときの調整も可能となる。かりにひとつのサーバーがダウンしたとしても、そこに割り当てられていた範囲を多数のサーバーに分散させることができ、ダウンしたサーバーの直近のサーバーだけに負荷をかけることはない。

13.5　整合性

これまでは、「データを複数マシンにレプリケートすれば永続化できるし負荷分散もできる」と、その利点だけを説明してきた。このあたりでその実態も書いておこう。複数マシンにデータのレプリカを置いてお互いの整合性を保つというのは、大変な作業だ。実際のところ、レプリカが壊れて同期できなくなることもあるだろうし壊れたデータを復旧できなくなることもあるだろう。ネットワーク上で複数のレプリカセットができてしまったり、マシン間のメッセージが遅延したり途中で消えてしまったりといったこともあり得る。NoSQL の世界でデータの整合性を保つための方法としてよく使われるのは、次の二通りの手法だ。ひとつは強整合性 (strong consistency) で、これはすべてのレプリカを同期させる。もうひとつが結果整合性 (eventual consistency) で、こちらの方式ではレプリカが同期されていなくてもかまわないが、最終的にはお互い相手側の状態に追いつけるようにしておかなければならない。まず最初に、どんな場合に結果整合性が選択肢に入るのかを分散コンピューティングの本質から考える。その後で、それぞれの手法の詳細を見ていこう。

CAP について

データに対する強整合性を保証できなくなることについて、なぜそんなに考慮するのだろう? すべては、今どきのネットワーク機器上で構築された分散システムの特性によるものだ。Eric Brewer が初めて提唱した *CAP* **定理**は、後に Gilbert と Lynch [GL02] によって証明された。この定理では、まず最初に分散システムの三つの特性を提示する。この頭文字をまとめた頭字語が CAP である。

整合性 (Consistency): あるデータのすべてのレプリカについて、どれを読んでも同じバージョンのデータを得られるか?(ここでいう整合性は、ACID の C とは異なる)

可用性 (Availability): アクセス不能なレプリカがいくつあっても、読み書きのリクエストに対応できるか?

耐分断性 (Partition tolerance): レプリカの一部が一時的にネットワーク上で他と分断されたときに、それでもシステムを稼働させ続けられるか?

そして、定理はさらにこう続く。複数台のコンピュータで構成されるストレージシステムは、この三つのうちの二つまでしか達成できず、その二つを達成するためには残りの一つが犠牲になってしまうのだ、と。また我々は、耐分断性を保証するシステムを実装せざるを得ない。現状のネットワーク機器やメッセージングプロトコルでは、パケットをロストすることもあればスイッチが故障することもある。そして、ネットワークがダウンしていたりメッセージの送信先のサーバーが存在しなかったりといったことを知るすべもない。すべての NoSQL システムで、耐分断性が必須となる。残された選択肢は、整合性と可用性のどちらを妥協するかである。両方を保証できるような NoSQL システムは存在しない。

整合性を保証するというのは、レプリカ間でのデータの同期をきちんと保つということだ。これを実現する簡単な方法は、すべてのレプリカに対する更新を確認することである。もしどれか一つのレプリカがダウンして更新確認を受け取れなかった場合は、そのキーの可用性を下げる。つまり、ダウンしたレプリカが復旧して確認の応答を返すまでは、その更新処理が成功したとは見なさないということだ。つまり、整合性を保証しようとすると、各データアイテムに対する 24 時間体制の可用性は保証できなくなる。

可用性を保証するというのは、ユーザーが何らかの操作を実行したときには、他のレプリカの状態がどうであるかにかかわらず自身が持つデータ上で操作を受け付けなければならないということだ。これは、レプリカ間でのデータの整合性を失ってしまうことにつながる。というのも、各レプリカにすべての更新が行き渡っているかどうかの確認がないので、中には一部の更新を受け取れていないレプリカも発生しうるからである。

CAP 定理からの帰結として、強整合性あるいは結果整合性のいずれかの手法で NoSQL データストアが作られることになった。それ以外の手法も存在して、たとえば Yahoo!の PNUTS [CRS+08] システムでは、緩い整合性と緩い可用性 (relaxed consistency and relaxed availability) という手法をとっている。しかし、本章で扱っているオープンソースの NoSQL システムの中にはまだこの手法を採用しているものが存在しないので、本章ではこの手法は扱わない。

強整合性

強整合性を謳うシステムが保証するのは、データ項目のレプリカが常にそのキーの値を保持しているということだ。レプリカの中には、他のレプリカとの同期がとれなくなっているものもあるかもしれない。しかしそんな場合であっても、たとえばユーザーが `employee30:salary`

の値を問い合わせれば、ユーザーに返す値は常に一貫性のあるものとなる。その原理を、数字で説明しよう。

ひとつのキーをN台のマシンにレプリケートしたものとする。あるマシン、おそらくN台のなかのどれかが、コーディネーターとしてユーザーからのリクエストを処理する。このコーディネーターは、N台のマシンのうちの一定台数以上が各リクエストを受け付けたことを保証する。あるキーに対する書き込みや更新があれば、少なくともW台のマシンがその更新を処理し終えたことを確認するまでコーディネーターはユーザーに応答を返さない。ユーザーが何らかのキーの値を読もうとしたときには、少なくともR台から同じ値を受け取るまでコーディネーターは応答を返さない。このとき、 R+W>N であればシステムの強整合性を実証できるものとする。

この考え方を、実際に数字を入れて確認しよう。各キーをN=3でレプリケートする(それぞれA、B、Cとする)。キー employee30:salary の初期値は$20,000だったが、ここで employee30 を$30,000に昇給させることになった。要件として、W=2 つまり A、B、C のうちで少なくとも2台が書き込みリクエストを受け付けることとする。このときAとBが書き込みリクエスト (employee30:salary, $30,000) を受け付けると、コーディネーターはユーザーに対して employee30:salary が更新できたと伝える。マシンCが仮に employee30:salary へのリクエストを受け取れなかったとしよう。つまり、マシンCの値は$20,000のままである。ここでコーディネーターがキー employee30:salary に対する読み込みリクエストを受け取ると、そのリクエストを3台のマシンすべてに送信する。

- もし仮に R=1 で最初に応答したのが C だったとしたら、結果は$20,000となってこの社員はあまりうれしくないだろう。
- しかし、もし R=2 にしておけばコーディネーターは最初に C の値を受け取っても A あるいは B からの二番目の応答を待ち続ける。どちらが先に来ても先ほどの C の値と食い違っているのでさらに待ち続け、最終的に三番目のマシンからの応答を受け取った時点で$30,000が多数派であることを確認できる。

したがって、この場合に強整合性を満たすには、R>=2 にして R+W>3 を満たす必要がある。

書き込みリクエストで W 台のレプリカから応答が返ってこなかったり、読み込みリクエストで一貫性のある結果が R 台以上のレプリカから得られなかったりした場合には何が起こるのだろう?コーディネーター側としては、最終的にタイムアウトを発生させてユーザーにエラーを返すこともできるし、要件を満たすまでずっと待ち続けることもできる。どちらにしても、そのリクエストについては少なくとも一定期間はアクセス不能とみなされる。

R と W をどのように設定するかによって、何台のマシンが不調になってもキーに対するさまざまな操作が可能になるかが決まる。たとえば書き込み操作はすべてのレプリカにきちんと反映させたいのなら、W=N とすることになる。この場合、どれか一台でもレプリカが応答しなければ、書き込みはハングしたり失敗したりする。よくある選択肢は R + W = N + 1 とするもので、こうすれば強整合性を維持するために最低限必要な稼働台数を最小に抑えられ

る。多くの強整合性システムは W=N そして R=1 という設定を選んでいる。そうすれば、同期に失敗したノードをどうするかを考えずに済むからである。

HBase は HDFS 上でストレージをレプリケートする。これは分散型のストレージ層だ。HDFS は強整合性を保証する。HDFS では、全 N 台 (通常は 2 あるいは 3) のレプリカに書き込み終えるまで書き込みは成功しない。つまり W = N である。読み込みはひとつのレプリカだけで応答できるので、R = 1 となる。大量の書き込みによるダウンを避けるため、ユーザーから各レプリカへのデータの転送は、非同期で並列処理される。すべてのレプリカがデータのコピーを受け取ったら、最後にシステム上のデータを新しいものに置き換える処理が行われる。これはアトミックな処理で、全レプリカの整合性を保ったものだ。

結果整合性

Dynamo ベースのシステムである Voldemort や Cassandra そして Riak などでは、ユーザーが必要に応じて N や R、W を指定できるようにしている。R + W <= N であってもかまわない。つまり、強整合性と結果整合性のどちらを達成するのかをユーザーが選択できるということだ。ユーザーが結果整合性を選択した場合、仮にプログラマーが強整合性を望んだとしても、W < N ならレプリカを安心して扱えない。レプリカ間での結果整合性を提供するには、システム側でさまざまなツールを使ってレプリカを最新状態に保つことになる。まずは、さまざまなシステムがどのようにしてデータが古くなったことを検出するのかを見ていこう。それから、レプリカを同期する方法を考える。そして最後に、同期プロセスを高速化するための Dynamo の影響を受けた方法をいくつか紹介する。

バージョニングと衝突

あるキーについて、二つのレプリカが違うバージョンの値を返す可能性がある以上、データのバージョン管理や衝突の検出が重要になる。Dynamo ベースのシステムで使っているバージョニング方式が**ベクタークロック**だ。ベクタークロックとは、各キーにベクターを割り当て、そこにレプリカのカウンタを含める方式である。たとえば、何らかのキーのレプリカを A と B そして C で扱うとする。このときベクタークロックには三つのエントリ (N_A, N_B, N_C) があり、その初期値は (0,0,0) となる。

レプリカ上でキーの値が変更されるたびに、ベクター内でそれに対応するカウンタが加算される。直前のバージョンが (39, 1, 5) だったときに B がキーの値を変更すると、ベクタークロックは (39, 2, 5) に書き換わる。別のレプリカ、たとえば C が B からそのキーのデータの更新を受け取るときには、B からのベクタークロックを自分のものと比較する。自分のベクタークロックカウンタのほうが B から受け取ったものよりも小さい場合、自分のデータは古いバージョンなのであるから B の内容で上書きできる。B より C のほうが大きいカウンタとその逆のカウンタが両方ある場合、たとえば (39, 2, 5) と (39, 1, 6) などのようになった場合は、矛盾する更新があったとサーバーが判断して衝突したとみなす。

衝突の解決

衝突の解決方法は、システムによってさまざまである。Dynamo の論文では、衝突の解決はストレージシステムを使うアプリケーション側に任せている。二つのバージョンのショッピングカートを一つにまとめるのはそれほど面倒ではないだろうが、共同作業で編集していた文書の複数バージョンをまとめる際の衝突は人間のレビューがないと解決できないだろう。Voldemort はこのモデルを採用しており、あるキーに対して複数のコピーを返し、クライアントアプリケーション側での対応を求める。

Cassandra は各キーのタイムスタンプを格納しており、二つのバージョンが衝突する場合は一番タイムスタンプの新しいバージョンを採用する。これによってクライアントとのやりとりの必要をなくし、API を単純化している。この設計では、衝突したデータを先ほどのショッピングカートの例のように自動マージすることは難しいし、分散カウンタを実装するのも困難だ。Riak は、Voldemort と Cassandra のどちらの手法でも使える。CouchDB はハイブリッド方式だ。衝突を検出したら、ユーザー側でそのキーを手動で修復させるよう問い合わせ、衝突が解決するまでは特定のバージョンを確定的に採用してユーザーに返すようになっている。

リードリペア

R 台のレプリカが衝突していないデータをコーディネーターに返せたら、コーディネーターはその衝突していない値を安全にアプリケーションに返せる。それでもコーディネーターは、同期ができていないレプリカがあることを検出するかもしれない。そんな場合に使えるテクニックとして Dynamo の論文で提案されており、Cassandra や Riak そして Voldemort が実装しているのが**リードリペア**だ。コーディネーターが読み込み時に衝突を検出すると、たとえ整合性のある結果をユーザーに返せたとしても、コーディネーターは衝突したレプリカの衝突解決プロトコルを開始する。これで、追加作業を最小限にしながら能動的に衝突を解消できる。各レプリカは既に自分の持つデータをコーディネーターに送っているので、早期に衝突を解決すればシステム内でのデータの相違を抑えられる。

Hinted Handoff

Cassandra や Riak そして Voldemort はすべて、*Hinted Handoff* というテクニックを使っている。これは、どれか一つのノードが一時的に使えなくなっている状態での書き込みのパフォーマンスを向上させるテクニックだ。あるキーに対応するレプリカの一つが書き込みリクエストに反応しなかったときに、別のノードを選択して一時的にその書き込み処理を引き継がせる。反応しなかったノードへの書き込みは個別に続けられ、反応しなかったノードが復旧したことをバックアップノードが知った時点で、そのレプリカに新しい書き込みをすべて転送する。Dynamo の論文では’sloppy quorum’ という手法を利用しており、Hinted Handoff で書き込まれたノードも書き込みの成功判断基準である W にカウントできるようにしている。

Cassandra や Voldemort は Hinted Handoff のぶんを W にカウントせず、本来割り当てられているレプリカの中で W に満たない場合は書き込みに失敗する。それでもなお Hinted Handoff は有用だ。というのも、反応しなくなったノードが復旧したときのリカバリーを高速に行えるからである。

Anti-Entropy

あるレプリカのダウン長期間になったり、ダウンしたレプリカを Hinted Handoff で引き継いだマシン自体もまたダウンしてしまった場合、レプリカはお互いに同期しなければならない。この場合に Cassandra や Riak が使う手順は、Dynamo に影響を受けた *Anti-Entropy* というものだ。Anti-Entropy において、各レプリカは**マークル木** *(Merkle Trees)* を交換する。これは、担当するキー範囲のうち最新状態と同期できていない部分を識別するものだ。マークル木とは、ハッシュの検証を階層的に行うものだ。もしキー空間全体のハッシュが二つのレプリカで一致しなければ、レプリケートしているキー空間のより小さい部分のハッシュを順に交換していき、同期できていないキーが特定できるまでそれを続ける。この手法により、ほとんど同じ状態で一部だけ違うというレプリカの間でのデータ交換の量を減らせる。

Gossip

分散システムが成長するにつれ、システム内の個々のノードが何をしているのかを追跡するのが難しくなってくる。Dynamo ベースの三つのシステムが他のノードを追跡するために使っているのは、*Gossip* という昔ながらのテクニックだ。定期的 (毎秒など) に、あるノードがランダムに別のノードを選んでお互いに通信し、自分が知っている他のノードの健康状態を交換する。このようにすることで各ノードは他のノードがダウンしているかどうかを知ることができ、クライアントからのキーの検索要求をどこに振ればいいかもわかるようになる。

13.6　最後に

NoSQL を取り巻く世界はまだ成熟しておらず、今回議論したシステムの多くもそのアーキテクチャや設計そしてインターフェイスを変えていくかもしれない。本章を読んで得られるものは、個々の NoSQL システムが現時点で何をしているかということではない。これらのシステムが、どのような決断を経て現在の機能セットに至ったのかということだ。NoSQL は、設計作業の多くをアプリケーション側の設計に委ねた。これらのシステムのアーキテクチャについて理解すれば、単に次世代のすばらしい NoSQL システムを作るにとどまらず、現時点のバージョンを責任を持って使えるようにする手助けになることだろう。

13.7　謝辞

Jackie Carter や Mihir Kedia、そして匿名のレビューアのみなさんに感謝する。みなさんのコメントや提案が本章の改善に大いに役立った。また本章は、NoSQL コミュニティの長年の作業がなければ存在し得なかった。これからもぜひこの調子で!

第14章

Python Packaging

Tarek Ziadé

14.1 はじめに

アプリケーションのインストールに関しては、ふたつの派閥がある。ひとつは、「アプリケーション単体で完結させる」派だ。Windows や Mac OS X で主流な考え方で、インストールのときに他の何かに依存するようではいけないというものだ。この思想に従えば、アプリケーションの管理はシンプルになる。各アプリケーションがそれ単体で動く “アプライアンス製品” で、インストールやアンインストールをしても OS には何ら影響を及ぼさない。もしそのアプリケーションが特殊なライブラリを使うのなら、ライブラリ自体もアプリケーションの配布物に含めることになる。

もう一方の派閥は「ソフトウェアとは、完結した小さな**パッケージ**の集合体である」派で、これは Linux ベースのシステムで主流である。ライブラリもパッケージとして同梱し、そのライブラリパッケージが他のパッケージに依存する可能性もある。何かのアプリケーションをインストールする際には、何十個もの他のライブラリについて特定のバージョンを探してインストールするということだってある。こういった依存情報の取得は中央リポジトリから取得することが多く、このリポジトリに何千ものパッケージがまとまっている。この思想があって、Linux ディストリビューションはそれぞれ複雑なパッケージ管理システム (dpkg や RPM など) を使っている。依存関係を追跡したり、同一ライブラリの互換性がない複数バージョン使うアプリケーションの共存を防いだりなどといったことを行う。

どちらの手法にだって、利点もあれば欠点もある。高度にモジュール化されたシステムで、すべてのパーツを個別に更新したり入れ替えたりできるようにしておけば、管理は楽になる。各ライブラリは一か所にしか存在せず、ライブラリを更新すればそれを使うすべてのアプリケーションがその恩恵を受けられるからである。たとえば、何かのライブラリにセキュリティフィックスを適用すれば、そのライブラリを使っているすべてのアプリケーションにその修正がいきわたる。一方、各アプリケーションがライブラリ群を同梱して出荷する形式の場合、セキュリティフィックスの適用がより複雑になる。特に、さまざまなアプリケーションが同じライブラリの異なるバージョンを使っている場合などが大変だ。

しかし、そんなモジュール方式も、人によっては欠点だと感じることがある。というのも、アプリケーションや依存関係をきちんと管理できなくなるからだ。そんな人たちにとっては、単体で完結したソフトウェア製品を提供して安定した環境を保証されるほうが安心する。システムの更新のときにも「依存地獄」に悩まされずに済むからである。

自己完結方式のアプリケーションは、複数の OS へのインストールに対応するのにも楽である。プロジェクトによっては、完全にポータブルなアプリケーションをリリースすることもある。導入先のシステムとの間のやりとりを**一切**せずに自分のディレクトリの中だけで動作するというものだ。ログファイルでさえもこのディレクトリの中で完結する。

Python のパッケージングシステムは、後者の思想 (インストールごとに複数の依存関係が生じるもの) に基づいて作られており、開発者にも管理者にもパッケージ作成者にも利用者にも可能な限り使いやすくしようとしている。残念ながらさまざまな不具合があって、考えうる限りのいろんな問題がこれまでに発生してきた (というか、今もなお発生している)。「バージョンスキーマがわかりにくい」「データファイルの処理を間違える」「パッケージの再作成が難しい」などなど。三年前、私を含む Python 開発者グループは、これらの課題に対応すべく新たな仕組みの開発に乗り出し、「パッケージングの仲間」を名乗った。[1] 本章では、私たちがこれまでに解決しようとした問題やどんなふうに問題を解決したのかを解説する。

用語

Python の世界では、**パッケージ**といえば Python ファイルを含むディレクトリのことである。Python ファイルのことは**モジュール**と呼ぶ。そんなこともあって、“パッケージ” という言葉を使うときには少し曖昧な感じになってしまう。というのも、多くのシステムではプロジェクトの**リリース**のこともパッケージと呼ぶからである。

Python 開発者ですら、ときどき混乱することがある。紛らわしさを排除する方法のひとつは、Python モジュールを含むディレクトリを指すときには「Python パッケージ」と言うようにすることだ。「リリース」は何かのプロジェクトのひとつのバージョンを表す用語として使い、あるリリースのソースあるいはバイナリを tar や zip でまとめたもののことを「ディストリビューション (配布物)」と呼ぶことにする。

14.2 Python 開発者の苦悩

Python プログラマーなら誰だって、自分のプログラムをいろんな環境で使えるようにしたいものだ。また、標準の Pythonr ライブラリとシステム依存のライブラリを組み合わせて使いたいことも多い。しかし、既存のすべてのパッケージングシステム用にそれぞれ個別のパッ

[1]訳注: “the Fellowship of the Packaging”。元ネタは『指輪物語』の “The Fellowship of the Ring” か。

ケージを作るのでもなければ、Python 用のリリース—つまり、OS が何であるかにかかわらず、Python をインストールした環境に導入することを狙いとしたリリース—を作って、後はこんなことを願うしかなくなる。

- すべての対象システムのパッケージ担当者が、個別のシステム用にパッケージを作り直せるようにする
- 依存関係も、すべての対象システムで再パッケージできるようにする
- システムの依存関係をきちんと表せる

そもそもそんなことは不可能だという場合もある。たとえば Plone(Python 製の本格的な CMS) は、ピュア Python の小さなライブラリを何百も使っている。いま出回っている各種パッケージングシステムで、これらのライブラリがすべてパッケージ化されているとは限らない。つまり Plone の場合は、必要なものをすべて同梱してそれ単体で機能するアプリケーションに**せざるを得ない**ということだ。そのために、Plone では `zc.buildout` を使っている。これは、すべての依存情報を収集してポータブルなアプリケーションを作り、そのディレクトリ単体だけであらゆるシステムで実行できるようにする仕組みだ。これは事実上、バイナリリリースに等しい。C のコードもすべてコンパイル済みだからである。

この方式は、開発者にとっては大勝利と言えるだろう。後述する Python の標準規格にしたがって依存関係を記述しておきさえすれば、あとは `zc.buildout` を使えばアプリケーションをリリースできるのだから。しかし、先述のとおり、この形式のリリースはシステムの中に要塞を構築するのに等しい。たいていの Linux システム管理者は、この方式を嫌うだろう。Windows の管理者は別に気にしないかもしれない。でも CentOS や Debian の管理者にとっては一大事だ。というのも彼らにとっては、システム上のすべてのファイルがベースとなる管理システムに登録されていることが前提であり、すべてのファイルを管理ツールで管理できることが前提だからである。

彼らは、あなたの作ったアプリケーションを自分たちの流儀に沿ってパッケージしなおそうとするかもしれない。ここで問われるのは「Python で、他のパッケージングシステムにも自動変換できるようなパッケージングシステムって作れないの?」ということだ。もしそれが実現できれば、特に追加作業をしなくてもひとつのアプリケーションやライブラリをあらゆるシステムにインストールできるようになる。ここで言う「自動化」は、必ずしもすべての作業をスクリプトで済ませるということではない。RPM や dpkg のパッケージを作っている人に聞いてみよう。そんなの不可能だってことがわかるはずだ。彼らはいつも、パッケージを作り直すときにちょっとした手作業が必要になっている。あと、こんなことも教えてくれるだろう。「何かコードをパッケージしなおすってのは大変なんだよ。だって、開発者ときたら、パッケージングのときのお約束なんてこれっぽっちも気にしないんだから」

既存の Python パッケージングシステムを使うパッケージャーを困らせる方法をひとつ教えよう。“MathUtils” という名前のライブラリの、バージョン “Fumanchu” リリースするんだ。どこかの賢い数学者がライブラリを書いたときに、「バージョン番号のかわりに、うちの飼い猫の名前を順番につけていこう。イケてるよね？」とでも思ったんでしょうなあ。実

は “Fumanchu”っていうのは二番目の猫の名前で、最初の猫は “Phil” だったんだ。つまり、“Fumanchu” は “Phil” の次のバージョンなんだけど、そんなことパッケージャーが知るわけがない。

いくらなんでも極端すぎると思うかもしれないが、いまどきのツールや標準規約を使っていると、十分起こり得ることだ。最悪なのは、easy_install や pip といったツールがそれぞれ非標準のレジストリを自前で持っており、インストールしたファイルをそこで管理しているってことだ。そこでは、“Fumanchu” や “Phil” といったバージョン名はアルファベット順に並んでしまう。

もうひとつ問題がある。データファイルの扱いをどうするかということだ。たとえば、もしあなたのアプリケーションで SQLite データベースを使うとしたらどうだろう? もしパッケージディレクトリの中にデータベースを置いたら、アプリケーションはうまく動かないかもしれない。アプリケーションを実行する人には、ツリー内のその部分に対する書き込み権限がないかもしれないからだ。それだけでなく、Linux システムでの前提も崩してしまう。Linux システム上では、バックアップ対象になるアプリケーションのデータを/var に置くことになっているのだ。

実際には、システム管理者がアプリケーションのファイルを好きなところに置けるようにする必要がある。それでアプリケーションが動かなくなることがあってはならず、アプリケーションのファイルがどこにあるかを管理者に伝えなければならない。そこで、先ほどの質問をこう言い換えてみよう。「Python で、こんなパッケージングシステムは実現可能だろうか? サードパーティのパッケージングシステムがパッケージを作り直すときに、わざわざアプリケーションのコードを追う必要がなく、みんなが幸せになれるようなシステムを」

14.3　現在のパッケージングのアーキテクチャ

Distutils パッケージが Python 標準ライブラリとして組み込まれている。これは、先述の問題たちに対応するために用意された。標準で組み込まれているので、問題はあるけれどもそれを使っているという人もいる。あるいは、もっとよくできた Setuptools などのツールを使う人もいる。これは Distutils に機能を追加したものだ。あるいは、Setuptools のフォークである Distribute という選択肢もある。さらに、もっと高機能なインストーラーとして Pip も存在する。これは Setuptools に依存している。

しかし、これらのツールはすべて Distutils をベースにしたものであり、その問題点も引き継いでいる。Distutils 自体を改善しようという動きもあったが、他のツールがあまりにもそのコードに依存しすぎていたので、内部のコードをほんの少し変更するだけでも Pythonp パッケージの世界を破滅させてしまう恐れがあった。

そこで私たちは、Distutils の開発を停止して、同じコードベースで Distutils2 の開発を新たに始めた。これで後方互換性を気にせずに済む。何を変えたのか、なぜ変えたのか。それを理解するために、まずは Distutils について詳しく見ていこう。

Distutils の基本とその設計ミス

Distutils に含まれるコマンド群はどれも、run メソッドを持つクラスである。いろんなオプションを指定して呼べる。Distutils では Distribution クラスも提供しており、ここに格納されているグローバルな値は全コマンドから参照できる。

Distutils を使いたい開発者は、自分のプロジェクトに Python モジュールをひとつ追加すればよい。慣例的に、名前は setup.py とすることになっている。このモジュールで、Distutils のエントリポイントである setup 関数を呼び出す。この関数にはいろんなオプションを指定できる。指定したオプションは、Distribution で保持してコマンドから使えるようになる。ひとつ例を示そう。ここでは、標準的なオプションをいくつか定義している。プロジェクトの名前とバージョン、そして含まれるモジュールの一覧だ。

```
from distutils.core import setup

setup(name='MyProject', version='1.0', py_modules=['mycode.py'])
```

このモジュールを使えば Distutils のコマンドを実行できる。たとえば sdist は、ソースディストリビューションをアーカイブして dist ディレクトリに置く。

```
$ python setup.py sdist
```

同じスクリプトを使って、install コマンドでこのプロジェクトをインストールすることもできる。

```
$ python setup.py install
```

Distutils には、それ以外にもこんなコマンドがある。

- upload…ディストリビューションをオンラインリポジトリにアップロードする。
- register…プロジェクトのメタデータをオンラインリポジトリに登録する。このときに必ずしもディストリビューションをアップロードする必要はない。
- bdist…バイナリディストリビューションを作成する。
- bdist_msi…Windows 用の.msi ファイルを作成する。

また、プロジェクトに関する情報をコマンドラインオプションで取得することもできる。

つまり、このプロジェクトをインストールしたり情報を調べたりといった作業は、ぜんぶこのファイルを使って Distutils 経由で行える。たとえば、プロジェクトの名前を調べるには次のようにする。

```
$ python setup.py --name
MyProject
```

つまり setup.py は、プロジェクトに関して何かをしたいときに常に間に入るものだということだ。ビルドからパッケージ作成、公開、インストールに至るまですべてである。開発者はプロジェクトの内容を関数へのオプションで指定し、また、このファイルを使ってパッケージ作成の作業もこなす。このファイルはまた、導入先のシステムでプロジェクトをインストールするときにも使われる。

たったひとつの Python モジュールでパッケージ作成やらリリースやら**挙句の果てには**インストールまでさせてしまう、というのが Distutils の問題のひとつだ。たとえば lxml プロジェクトの名前を取得したいとしよう。setup.py は、単に名前を文字列で返す以外にもいろんなことをする。

```
$ python setup.py --name
Building lxml version 2.2.
NOTE: Trying to build without Cython, pre-generated 'src/lxml/lxml.etree.c'
needs to be available.
Using build configuration of libxslt 1.1.26
Building against libxml2/libxslt in the following directory: /usr/lib/lxml
```

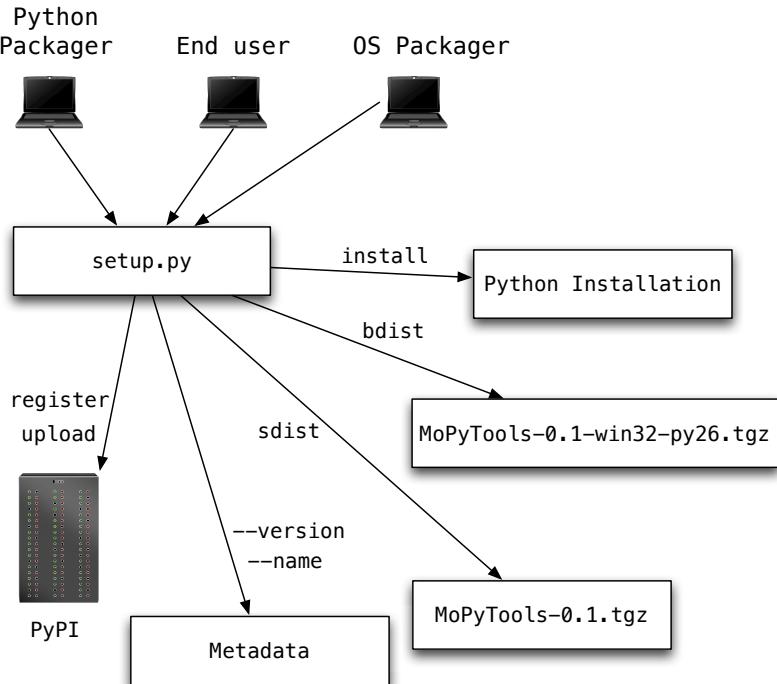



図 14.1: Setup

場合によってはこれがうまく動かないかもしれない。「setup.py なんて、インストールのときにしか使わないよね」と思いこんだ開発者が、それ以外の Distutils の機能をユーザーが使うとは想定していないような場合がある。setup.py にいろんな役割を持たせてしまうのは混乱のもとだ。

Metadata と PyPI

Distutils は、ディストリビューションを作るときに Metadata ファイルも作る。これは、PEP 314 [2] で指定された標準規約に従うものである。その中に含まれるのは通常のメタデータ、たとえばプロジェクト名やリリースのバージョンなどである。主要なフィールドを以下に示す。

- Name: プロジェクト名。
- Version: リリースのバージョン。
- Summary: 一行にまとめた説明。
- Description: 詳しい説明。

[2] Python Enhancement Proposals(本章では PEP と略する) については、本章の最後にまとめる

- Home-Page: プロジェクトの URL。
- Author: 作者名。
- Classifiers: プロジェクトの分類。使える分類の一覧は Python が用意しており、たとえばライセンスやリリースの成熟度 (beta、alpha、final) などを指定できる。
- Requires、Provides、Obsoletes: モジュールの依存関係。

これらの項目の大半は、他のパッケージシステムにも同様なものが用意されているので簡単に対応できる。

Python Package Index (PyPI)[3] は、CPAN のようにパッケージを取りまとめた中央リポジトリである。ここにプロジェクトを登録したりリリースを公開したりするには Distutils の register コマンドや upload コマンドを使えばよい。register は、Metadata ファイルを構築して PyPI に送信する。これで、利用者やツール (インストーラなど) がそのパッケージを (ウェブページやウェブサービス経由で) 見つけられるようになる。

図 14.2: PyPI リポジトリ

Classifiers にもとづいてプロジェクトを閲覧し、その作者名やプロジェクトの URL を取得することができる。一方、Requires を使えば Python モジュールとの依存関係を定義できる。requires オプションを使って、プロジェクトに Requires メタデータ要素を追加できる。

```
from distutils.core import setup

setup(name='foo', version='1.0', requires=['ldap'])
```

ldap モジュールに対する依存関係の定義は純粋に宣言的なものであり、インストーラーやツールでそのモジュールの存在を保証できるわけではない。Python が依存関係をモジュール

[3]かつて CheeseShop と呼ばれていたもの。

レベルで定義できるのなら、これでも十分だ。そう、ちょうど Perl の require キーワードみたいにね。そうすれば、インストーラーは単に PyPI で依存情報を確認してそれをインストールするだけになる。CPAN がやっているのは、基本的にそういうことだ。でも Python ではそれができない。というのも、ldap という名前のモジュールはあらゆる Python プロジェクトに存在する可能性があるからである。Distutils では複数のパッケージやモジュールを含むプロジェクトをリリースできるので、このメタデータフィールドはまったく使い物にならない。

Metadata ファイルには、それ以外の問題もある。このファイルは Python スクリプトで作られるので、スクリプトを実行したプラットフォーム専用のものになってしまうのだ。たとえば、Windows 専用の機能を提供するプロジェクトでは、setup.py で次のような定義をすることになるだろう。

```
from distutils.core import setup

setup(name='foo', version='1.0', requires=['win32com'])
```

しかしこれでは、たとえ実際には他のプラットフォームでも同じ機能が使える場合でも、このプロジェクトは Windows でしか動かないということになってしまう。ひとつの解決策として、Windows の場合だけ requires オプションを指定するという方法もある。

```
from distutils.core import setup
import sys

if sys.platform == 'win32':
    setup(name='foo', version='1.0', requires=['win32com'])
else:
    setup(name='foo', version='1.0')
```

しかし、そんなことをすれば問題はさらに悪化する。思い出してみよう。このスクリプトは、PyPI 経由で世に送り出すためのソースアーカイブを作るときにも使うものだった。つまり、PyPI に送られる Metadata ファイルは、コンパイルしたプラットフォームに依存するものになってしまう。つまり、プラットフォームによって値が異なるようなメタデータフィールドを用意するのは不可能だということである。

PyPI のアーキテクチャ

先述のとおり、PyPI は Python のプロジェクトをまとめた索引である。カテゴリを指定してプロジェクトを探したり、自分のプロジェクトを登録したりできる。ソースディストリビューションやバイナリディストリビューションをアップロードしてプロジェクトに追加すれば、それを他の人がダウンロードしてインストールできるようになる。PyPI はウェブサービスも提供しており、インストーラなどのツールはこのサービスを利用する。

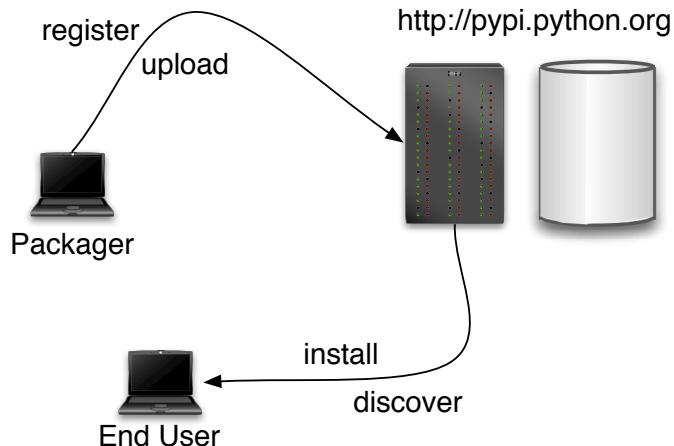



図 14.3: PyPI のワークフロー

プロジェクトの登録とディストリビューションのアップロード

プロジェクトを PyPI に登録するには、`Distutils` の `register` コマンドを使う。このコマンドは、プロジェクトのメタデータやバージョンを含む POST リクエストを作る。このリクエストには Authorization ヘッダが必要となる。というのも PyPI は、最初にプロジェクトを登録した人と同一ユーザーであるかどうかを確認するためにベーシック認証を使っているからである。認証情報はローカルの `Distutils` の設定に保存することもできるし、`register` コマンドを実行するたびに入力することもできる。使い方は、次のようになる。

```
$ python setup.py register
running register
Registering MPTools to http://pypi.python.org/pypi
Server response (200): OK
```

プロジェクトを登録すると、そのプロジェクト用のウェブページにメタデータの HTML バージョンが表示される。ディストリビューションを PyPI にアップロードするには `upload` を使う。

```
$ python setup.py sdist upload
running sdist
...
running upload
Submitting dist/mopytools-0.1.tar.gz to http://pypi.python.org/pypi
Server response (200): OK
```

ファイルを直接 PyPI にアップロードするかわりに、別の場所を使うよう指定することもできる。その場合は、メタデータの `Download-URL` フィールドを指定する。

PyPI への問い合わせ

PyPI は、ウェブ経由での利用者向けに HTML ページを作るだけでなく、ふたつのサービスも提供する。Simple Index プロトコルと XML-RPC API だ。

Simple Index プロトコルは `http://pypi.python.org/simple/` から始まるプレーンな HTML ページで、登録されている全プロジェクトへの相対リンクを含む。

```
<html><head><title>Simple Index</title></head><body>
...
<a href='MontyLingua/'>MontyLingua</a><br/>
<a href='mootiro_web/'>mootiro_web</a><br/>
<a href='Mopidy/'>Mopidy</a><br/>
<a href='mopowg/'>mopowg</a><br/>
<a href='MOPPY/'>MOPPY</a><br/>
<a href='MPTools/'>MPTools</a><br/>
<a href='morbid/'>morbid</a><br/>
<a href='Morelia/'>Morelia</a><br/>
<a href='morse/'>morse</a><br/>
...
</body></html>
```

たとえば MPTools プロジェクトには `MPTools/` というリンクがある。これは、そのプロジェクトがインデックスに存在することを意味する。リンク先のサイトに、そのプロジェクトに関連するすべてのリンクが含まれる。

- PyPI に格納されているすべてのディストリビューションへのリンク
- 登録されている各バージョンについての、`Metadata` で定義されたすべてのホーム URL
- 同じく登録されている各バージョンについての、`Metadata` で定義されたすべてのダウンロード URL

MPTools のページの内容は、このようになる。

```
<html><head><title>Links for MPTools</title></head>
<body><h1>Links for MPTools</h1>
<a href="../../packages/source/M/MPTools/MPTools-0.1.tar.gz">MPTools-0.1.tar.gz</a><br/>
<a href="http://bitbucket.org/tarek/mopytools" rel="homepage">0.1 home_page</a><br/>
</body></html>
```

インストーラなどのツールがプロジェクトのディストリビューションを探したいときは、インデックスページを調べるか、あるいは単に `http://pypi.python.org/simple/PROJECT_NAME/` が存在するかどうかを調べればよい。

このプロトコルには、ふたつの大きな制限がある。まず第一に、PyPI は現時点では単一のサーバーで運用しているということ。多くの人はそのコンテンツをローカルにコピーしているが、過去二年間にも何度か PyPI がダウンすることがあった。そのたびに、PyPI を見てプロジェクトの依存関係を解決するインストーラが使えなくなって開発者を混乱させていた。たとえば、Plone アプリケーションをビルドするときには PyPI への問い合わせが数百件単位

で発生する。必要なパッケージを取得するためである。つまり、PyPI が単一障害点となる可能性があるわけだ。

そして第二に、ディストリビューションを PyPI に格納せずに Simple Index ページで Download-URL のリンクを提供している場合、インストーラはそのリンクをたどることになる。その場所が正常稼働中なのか、そこにブツが本当にあるのかを気にしながら。こういった回り道が、Simple Index ベースのすべての処理の弱点となる。

Simple Index プロトコルの目標は、あるプロジェクトのインストールに使うリンクの一覧をインストーラに渡すことだ。そのため、プロジェクトのメタデータは公開されていない。そのかわりに XML-RPC を使って、登録したプロジェクトに関する追加情報を取得できる。

```
>>> import xmlrpclib
>>> import pprint
>>> client = xmlrpclib.ServerProxy('http://pypi.python.org/pypi')
>>> client.package_releases('MPTools')
['0.1']
>>> pprint.pprint(client.release_urls('MPTools', '0.1'))
[{'comment_text': '',
'downloads': 28,
'filename': 'MPTools-0.1.tar.gz',
'has_sig': False,
'md5_digest': '6b06752d62c4bffe1fb65cd5c9b7111a',
'packagetype': 'sdist',
'python_version': 'source',
'size': 3684,
'upload_time': <DateTime '20110204T09:37:12' at f4da28>,
'url': 'http://pypi.python.org/packages/source/M/MPTools/MPTools-0.1.tar.gz'}]
>>> pprint.pprint(client.release_data('MPTools', '0.1'))
{'author': 'Tarek Ziade',
'author_email': 'tarek@mozilla.com',
'classifiers': [],
'description': 'UNKNOWN',
'download_url': 'UNKNOWN',
'home_page': 'http://bitbucket.org/tarek/mopytools',
'keywords': None,
'license': 'UNKNOWN',
'maintainer': None,
'maintainer_email': None,
'name': 'MPTools',
'package_url': 'http://pypi.python.org/pypi/MPTools',
'platform': 'UNKNOWN',
'release_url': 'http://pypi.python.org/pypi/MPTools/0.1',
'requires_python': None,
'stable_version': None,
'summary': 'Set of tools to build Mozilla Services apps',
'version': '0.1'}
```

この手法の問題は、XML-RPC API で公開しているデータの中には、静的ファイルとして格納して Simple Index ページで公開することもできるものがあるという点だ。そのほうが、ク

ライアントツールの作業量を減らせる。また、問い合わせに対応する PyPI 側としても、そうしておけば余計な手間を省ける。公開したディストリビューションごとのダウンロード数など、変化するデータはウェブサービスで公開するのに適している。しかし、プロジェクトに関する静的なデータを二種類の異なる方法で公開するのは意味がない。

Python Installation のアーキテクチャ

Python のプロジェクトを `python setup.py install` でインストールすると、標準ライブラリの `Distutils` がファイルをシステム上にコピーする。

- *Python* **パッケージ**やモジュールは Python のディレクトリに配置され、インタプリタの起動時に読み込まれる。最新の Ubuntu なら`/usr/local/lib/python2.6/distpackages/`だし、Fedora なら`/usr/local/lib/python2.6/sites-packages/`だ。
- プロジェクトで定義している**データファイル**はシステム上のどこにでも置ける。
- **実行可能スクリプト**は、システム上の `bin` ディレクトリに配置する。その場所はプラットフォームによって異なり、`/usr/local/bin` になるかもしれないし、Python のインストール先にある `bin` ディレクトリかもしれない。

Python 2.5 以降は、メタデータファイルもモジュールやパッケージとともにコピーされるようになった。ファイル名は `projectversion.egg-info` となる。たとえば `virtualenv` プロジェクトの場合は `virtualenv-1.4.9.egginfo` のようになる。このメタデータファイルは、インストール済みのプロジェクトについてのデータベースと考えられる。というのも、メタデータファイルを順に読んでいけばプロジェクトとそのバージョンの一覧を作れるからである。しかし、`Distutils` のインストーラはシステム上にインストールしたファイルの一覧をどこにも記録しない。つまり、いったんコピーしたファイルをアンインストールしようと思ってもできないということになる。情けない話だね。`install` コマンドには`--record`っていうオプションがあって、これを使えばインストールしたファイルの一覧をテキストファイルに残せるのに。このオプションは標準では使われていないし、`Distutils` のドキュメントにもほとんど説明がない。

Setuptools や Pip など

最初に説明したとおり、`Distutils` の問題点を解決しようというプロジェクトがいくつか存在する。それにどの程度成功したかは、プロジェクトによってさまざまだ。

依存関係の問題

PyPI では、複数のモジュールをひとつの Python パッケージにまとめたプロジェクトを公開できる。一方、あるプロジェクトに対してモジュールレベルの依存関係を `Require` で定義

することもできる。どちらも合理的だが、二つの方法を組み合わせるのはよくない。「まぜるな危険」だ。

ほんとうにやるべきことは、プロジェクトレベルの依存関係を定義することだった。まさにそれをやろうとしたのが Setuptools で、これは Distutils にプロジェクトレベルの依存管理機能を追加したものである。easy_install というスクリプトも付属し、PyPI 経由で自動的に依存関係を取得してインストールできるようにした。実際のところ、モジュールレベルの依存関係が使われることはなく、人はみな Setuptools の拡張機能に飛びついた。しかし、所詮は Setuptools でしか使えないオプションであり、Distutils や PyPI には無視されていた。事実上、Setuptools は独自規格を作ってしまったということになる。もともとイケてない設計のところに無理やり手を入れて、ね。

easy_install はどうしているのかというと、まずプロジェクトのアーカイブをダウンロードし、その setup.py を実行してメタデータを取得する。そして、その情報を見て依存関係を取得する。このように、依存グラフを少しずつ取得してはダウンロードをするという流れになる。

新しいメタデータが PyPI に登録されてオンラインで見えるようになっても、easy_install はアーカイブ全体をダウンロードする必要がある。というのも、先述のとおり、PyPI に公開されるメタデータはそれをアップロードしたプラットフォーム専用のものであり、インストール先のプラットフォームとは異なる可能性があるからだ。しかし、あるプロジェクトとその依存関係をインストールできるという機能はたいていの場合は必要十分なものだったし、あるべき機能だった。そのため Setuptools は広くつかわれるようになったが、まだ問題は残っていた。

- 依存パッケージのインストールに失敗したときにロールバックする手段がなく、システムが中途半端な状態になってしまう可能性がある。
- インストールを進めながらその場で徐々に依存グラフを構築していくので、途中で依存関係の衝突が見つかったらそこでインストールが止まってしまう。このときも、システムが中途半端な状態になってしまう。

アンインストールの問題

Setuptools にはアンインストーラが存在しない。インストールするファイルの一覧をカスタムメタデータとして持たせることもできるというのに。一方 Pip は、Setuptools のメタデータを拡張してインストールしたファイルを記録できるようして、アンインストールができるようになった。しかし、これまた Metadata の亜種である。ということは、Python のプロジェクトをインストールすると、ひとつのプロジェクトに対して最大四種類のメタデータが存在することになってしまう。

- Distutils の egg-info。これは単一のメタデータファイル。

- Setuptools の egg-info。これはディレクトリで、メタデータのほかに Setuptools 専用のオプションが追加されている。
- Pip の egg-info。これは Setuptools のメタデータをさらに拡張したもの。
- インストール先のシステムのパッケージ管理システムが作るもの。

データファイルとは?

Distutils では、データファイルはシステム上の好きなところにインストールできる。何かのパッケージのデータファイルを setup.py で次のように定義したとしよう。

```
setup(...,
  packages=['mypkg'],
  package_dir={'mypkg': 'src/mypkg'},
  package_data={'mypkg': ['data/*.dat']},
  )
```

このとき、mypkg プロジェクトで拡張子が.dat のファイルがすべてディストリビューションに含まれ、インストール時には Python モジュールと同じ場所にインストールされる。

Python ディストリビューションとは別の場所にインストールする必要のあるデータファイル用には、別のオプションがある。アーカイブ内のファイルを、指定した場所に配置するためのオプションだ。

```
setup(...,
    data_files=[('bitmaps', ['bm/b1.gif', 'bm/b2.gif']),
                ('config', ['cfg/data.cfg']),
                ('/etc/init.d', ['init-script'])]
    )
```

これ、OS のパッケージャにとっては悪夢のように聞こえるだろう。だって、

- データファイルはメタデータに含まれないので、パッケージャは setup.py を読んだりプロジェクトのコードを追ったりしなければいけない。
- 個々のシステムでデータファイルをどこに置くかなんて、開発者に決めさせることじゃないよね。
- データファイルをカテゴリ分けできない。画像も man ページもその他のデータも、みんな同じ扱いになってしまう。

そんなファイルを含むプロジェクトを再パッケージしようとしたパッケージャは、setup.py ファイルに手を入れて自分のプラットフォームでも動くようにする以外の道がない。setup.py ファイルに手を入れるには、コードを追ってデータファイルを使っているすべての場所を変更しないといけない。というのも、開発者はデータがその場所にあるという前提でコードを書いているからである。Setuptools や Pip を使ったところで状況は変わらない。

14.4　標準規約の改良

　というわけで、いろんなパッケージング環境が混在するややこしい状態を何とかすることにした。ひとつの Python モジュールですべての作業をこなしたり、メタデータが不完全だったり、プロジェクトの内容を完全に記述する方法がなかったり、そんなのはもういやだ。そこで私たちが何をどう改善したのかをまとめた。

メタデータ

　まず最初に取り組んだのは、`Metadata` の標準規格を修正することだった。PEP 345 で定義した新しいバージョンには以下の内容が含まれる。

- まともな方法でのバージョン定義
- プロジェクトレベルの依存関係
- プラットフォーム固有の値を静的に定義する方法

バージョン

　メタデータの標準化のひとつの目標は、Python プロジェクトを扱うすべてのツールが同じ方法でメタデータを扱えるようにすることだ。バージョン番号に関して言うと、バージョン “1.1” のほうが “1.0” より新しいということをすべてのツールがわかるようにしなければいけない。でも、プロジェクトによってバージョンの表し方が異なると、とたんにこの目標は困難になる。

　バージョンの表し方を統一する唯一の方法は、標準規格を公開してそれに従わせることだ。私たちが選んだのは、昔ながらの連番ベースの方式だった。PEP 386 で定義したとおり、その書式は次のようになる。

```
N.N[.N]+[{a|b|c|rc}N[.N]+][.postN][.devN]
```

ここで、

- *N* は整数値。ドットで区切って好きなだけ N をつなげることができる。ただし、少なくともふたつ (MAJOR.MINOR) 以上は必要。
- *a*、*b*、*c* および *rc* は、それぞれ *alpha*、*beta*、**リリース候補**を表すマーク。その後に数字を続ける。リリース候補を表すマークが *c* と *rc* の二種類あるのは、Python 自身が *rc* を使っていることに合わせたものである。私たちは、*c* のほうがよりシンプルだと考える。
- *dev* の後に数字を続けて、開発版を表す。
- *post* の後に数字を続けて、ポストリリース版を表す。

プロジェクトのリリース手順によっては、最終版の二つのリリースの間にある中間バージョンに対して dev あるいは post を使える。たいていのプロジェクトでは dev を使っている。

この方針に則って、PEP 386 では厳密な順番を定義している。

- alpha < beta < rc < final
- dev < non-dev < post、ここで non-dev とは alpha、beta、rc あるいは final を指す

例を示そう。

```
1.0a1 < 1.0a2.dev456 < 1.0a2 < 1.0a2.1.dev456
  < 1.0a2.1 < 1.0b1.dev456 < 1.0b2 < 1.0b2.post345
    < 1.0c1.dev456 < 1.0c1 < 1.0.dev456 < 1.0
      < 1.0.post456.dev34 < 1.0.post456
```

この方針の目標は、他のパッケージングシステムが Python のプロジェクトのバージョンを自分たちの管理する方式に変換しやすくすることだ。今や PyPI は、PEP 345 のメタデータをアップロードしたときに PEP 386 に従わないバージョン番号をつけているプロジェクトを拒否するようになった。

依存関係

PEP 345 では新たに三種類のフィールドを定義した。PEP 314 の `Requires`、`Provides` そして `Obsoletes` に取って代わるものだ。その名は `Requires-Dist`、`Provides-Dist` そして `Obsoletes-Dist` で、メタデータの中で複数回使える。

`Requires-Dist` では、このディストリビューションが必要とする他の `Distutils` プロジェクトの名前を文字列で指定する。文字列の書式は `Distutils` のプロジェクト名 (つまり `Name` フィールドにあるもの) と同じで、その後に続けて括弧内にバージョンを宣言できる。`Distutils` のプロジェクト名は PyPI で探すときの名前に対応し、バージョンの宣言方法は PEP 386 のルールに従う。いくつか例を示す。

```
Requires-Dist: pkginfo
Requires-Dist: PasteDeploy
Requires-Dist: zope.interface (>3.5.0)
```

`Provides-Dist` は、このプロジェクトに含まれる追加の名前を定義する。あるプロジェクトを別のプロジェクトと混ぜたい場合に有用だ。たとえば ZODB プロジェクトには `transaction` プロジェクトを含めることができ、その場合は次のように宣言する。

```
Provides-Dist: transaction
```

`Obsoletes-Dist` は、別のプロジェクトを非推奨扱いにできる。

```
Obsoletes-Dist: OldName
```

環境マーカー

環境マーカーは、フィールドの最後にセミコロンに続けて追加するマークで、実行環境に関する条件を指定する。いくつか例を示す。

```
Requires-Dist: pywin32 (>1.0); sys.platform == 'win32'
Obsoletes-Dist: pywin31; sys.platform == 'win32'
Requires-Dist: foo (1,!=1.3); platform.machine == 'i386'
Requires-Dist: bar; python_version == '2.4' or python_version == '2.5'
Requires-External: libxslt; 'linux' in sys.platform
```

環境マーカー用のマイクロ言語は、意図的にシンプルにしている。Python プログラマー以外にも理解してもらえるようにするためだ。文字列は==演算子と in 演算子 (そしてそれぞれの逆の演算子) で比較し、通常の論理演算子が使える。PEP 345 のフィールドの中でこのマーカーが使えるものは、次のとおりである。

- Requires-Python
- Requires-External
- Requires-Dist
- Provides-Dist
- Obsoletes-Dist
- Classifier

何がインストールされたのか?

単一のインストール形式を、すべての Python ツール群で共有する。相互運用性を確保するためには、これが必須である。たとえば、インストーラー B が以前にプロジェクト Foo をインストールしたという事実をインストーラー A で検出できるようにしたければ、インストールされたプロジェクトの情報を同じデータベースで共有しなければいけない。

もちろん、あれこれ使わずにひとつのインストーラーで揃えるのが理想ではある。でも、目新しい機能を搭載した新しいインストーラーが登場したら、つい使ってしまいたくなるものだ。たとえば、Mac OS X には標準で Setuptools が含まれているので、ユーザーは自動的に easy_install スクリプトを使えるようになる。ユーザーが新しいツールを使おうと思ったら、以前のツールとの互換性が必要となる。

別の問題もある。RPM などのパッケージ管理システムを持つプラットフォームで Python インストーラーを使ったときに、何かのプロジェクトがインストールされたことをシステム側に伝える方法がないのだ。さらに悪いことに、たとえ Python インストーラー側から何らかの方法でシステムのパッケージ管理システムに通知できたとしても、Python のメタデータとシステムのメタデータの間でマッピングが必要となる。たとえばプロジェクト名ひとつとってもお互い違うこともある。その理由はいくつかあって、いちばんありがちなのは名前の衝突だ。Python 界とは無関係の別のプロジェクトが、そのプロジェクトと同じ名前の RPM を

作っている場合などである。それ以外にあり得るのは、python のプレフィックスを含む名前がプラットフォーム側の命名規約と衝突してしまうという例だ。たとえば foo-python というプロジェクトを作ったとしよう。Fedora の RPM なら、高い確率でそれは python-foo という名前になるだろう。

この問題を回避する方法のひとつは、システム全体で使う Python の環境はシステムのパッケージングシステムにまかせておいて、それとは別に隔離した環境で作業することだ。Virtualenv のようなツールを使えば、それができる。

いずれにせよ、Python の世界でのインストーラーのフォーマットを統一する必要がある。だって、他のパッケージングシステムが Python のプロジェクトをインストールするときにも、相互運用性は重要になるからだ。サードパーティのパッケージングシステムが新しい Python プロジェクトをインストールして自前のデータベースに登録したら、何とかして Python 自身に正しいメタデータを伝える必要がある。そうすれば、Python インストーラー (あるいは Python プロジェクトのインストール状態を調べる何らかの API) からもそのプロジェクトがインストールされたことがわかるようになる。

メタデータのマッピングができれば、こんなときにも対応できる。RPM は自身が管理する Python プロジェクトを知っているので、そこから適切な Python メタデータを生成できるようになるのだ。たとえば、python26-webob は PyPI の世界では WebOb と呼ばれているといったマッピングだ。

さて、標準化の話題に戻る。PEP 376 では、インストール済みパッケージの標準フォーマットを定義した。Setuptools や Pip で使っているフォーマットと似た形式である。拡張子 dist-info のディレクトリに、このような内容を含める。

- METADATA: PEP 345、PEP 314 および PEP 241 で記されたメタデータ。
- RECORD: インストールされたファイルの一覧を csv 風の書式で表したもの。
- INSTALLER: このプロジェクトをインストールするときに使ったツールの名前。
- REQUESTED: このファイルが存在すれば、そのプロジェクトは明示的にインストールされたものだということを表す (依存関係を満たすために自動的にインストールされた場合は、このファイルは存在しない)。

すべてのツールがこのフォーマットに対応するようになれば、特定のインストーラーやその機能に頼らずに Python のプロジェクトを管理できるようになる。また、PEP 376 ではメタデータをディレクトリとして定義しているので、メタデータを拡張するときにも単に新しいファイルを追加するだけで済む。実際、次のセクションで解説する新たなメタデータファイル RESOURCES は、近い将来に追加される予定だ。これを追加しても PEP 376 には手を加えない。最終的にこのファイルがすべてのツールにとって有用だとわかれば、そのときに PEP に追加されることだろう。

データファイルのアーキテクチャ

先述の通り、インストール時のデータファイルの置き場所はパッケージャーが自由に決められるようにする必要がある。そして、どこに置いても開発者の書いたコードが動くようにもしなければいけない。同時に、開発者も、データファイルの置き場所がどこであるかを意識せずに作業できるようでなければいけない。そこで私たちがどうしたか。よくある手段、つまり間接参照だ。

データファイルの利用

MPTools アプリケーションで設定ファイルが必要になったとする。開発者は設定ファイルを Python パッケージの中に置き、`__file__`で参照しようとするだろう。

```
import os

here = os.path.dirname(__file__)
cfg = open(os.path.join(here, 'config', 'mopy.cfg'))
```

これは、設定ファイルがコードと同じようにインストールされており、かつ開発者が設定ファイルをコードと同じ場所に**置かないといけない**ということを前提としている。この例では config というディレクトリがあるものと見なしている。

データファイルの新たなアーキテクチャを設計する際には、このように考えた。プロジェクトツリーをすべてのファイルのルートとして使い、ツリー内のすべてのファイルに対して、Python パッケージであっても単純なディレクトリであってもアクセスできるようにした。これで、開発者はデータファイル専用のディレクトリを作れるようになり、そこに pkgutil.open でアクセスできるようになったのだ。

```
import os
import pkgutil

# Open the file located in config/mopy.cfg in the MPTools project
cfg = pkgutil.open('MPTools', 'config/mopy.cfg')
```

pkgutil.open は、プロジェクトのメタデータを読んで RESOURCES ファイルが含まれているかどうかを調べる。このファイルは、システムに含まれるファイルの場所をまとめたシンプルなマップだ。

```
config/mopy.cfg {confdir}/{distribution.name}
```

ここで、confdir が指すのはシステムの設定ディレクトリであり、distribution.name に含まれるのはメタデータに書かれた Python プロジェクト名である。

インストール時に RESOURCES ファイルを作っておきさえすれば、この API が mopy.cfg の場所を見つけてくれるようになる。また、config/mopy.cfg はプロジェクトのツリーからの相対パスなので、開発モードも用意できる。開発モードのときにはプロジェクトのメタデータをその場で生成し、pkgutil 用の検索パスを追加すればよい。

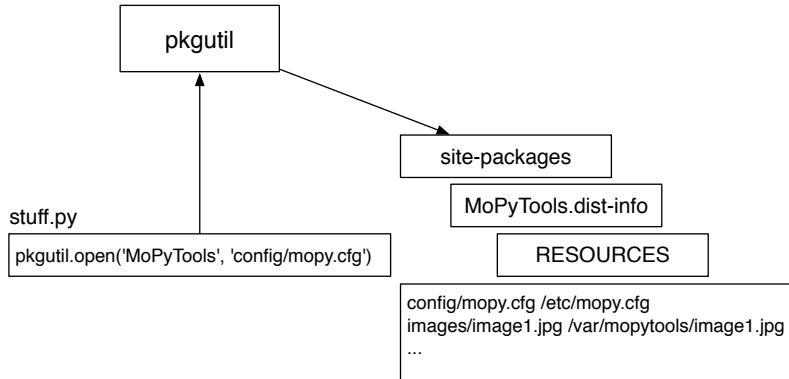



図 14.4: ファイルを探す流れ

データファイルの宣言

実際には、データファイルをどこに配置するのかはプロジェクトで定義できる。`setup.cfg` ファイルでマッパーを定義すればよい。マッパーというのは、`(glob-style pattern, target)` からなるタプルのリストである。各パターンがプロジェクトツリー内のいずれかのファイルを指し、ターゲットにはインストール先のパスを指定する。このパスには、ブラケットで囲んだ変数を含めることができる。たとえば `MPTools` の `setup.cfg` は次のようになる。

```
[files]
resources =
        config/mopy.cfg {confdir}/{application.name}/
        images/*.jpg    {datadir}/{application.name}/
```

`sysconfig` モジュールが、ここで使える変数の一覧やそのドキュメントを提供しており、各プラットフォームでのデフォルト値も設定している。たとえば、Linux 上での `confdir` のデフォルトは`/etc` となる。インストーラーは、このマッパーと `sysconfig` を組み合わせればファイルの配置先を知ることができる。最終的には先述の `RESOURCES` ファイルを生成してメタデータに含めるので、`pkgutil` が後からデータファイルを見つけられるようになる。

PyPI の改良

先述のとおり、PyPI は事実上の単一障害点だった。PEP 380 では、この問題に対応するためにミラーリングプロトコルを導入し、PyPI が落ちているときに他の代替サーバーにフォールバックできるようにした。その狙いは、コミュニティのメンバーが世界中でミラーを稼働させられるようにすることだ。

ミラーリストの形式は、`X.pypi.python.org` 形式のホスト名のリストとなる。ここで `X` は、`a,b,c,...,aa,ab,...` のようなシーケンスである。`a.pypi.python.org` がマスターサーバー

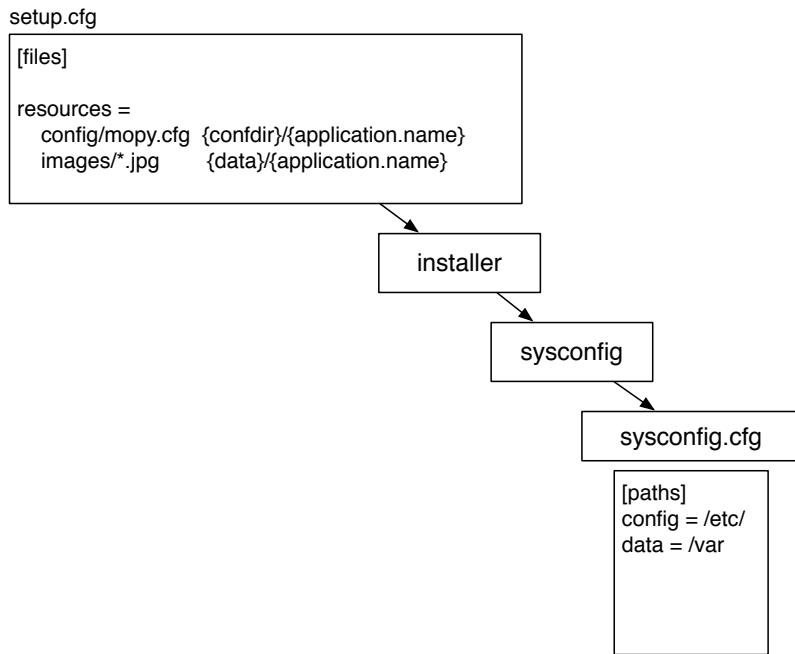



図 14.5: インストーラー

で、ミラーは b から始まる。CNAME レコード `last.pypi.python.org` は最後のホスト名を指しているので、PyPI を使うクライアントは、この CNAME を見ればミラー一覧を取得できる。

たとえば次の結果を見ると、最後のミラーが `h.pypi.python.org` であることがわかる。つまり、この時点で PyPI には六つのミラー (b から h) があるということだ。

```
>>> import socket
>>> socket.gethostbyname_ex('last.pypi.python.org')[0]
'h.pypi.python.org'
```

将来的に、このプロトコルを使ってクライアントから一番近いミラーへとリクエストをリダイレクトできるようにする可能性もある。IP アドレスを元に近場のミラーを探し、もしそのミラーあるいはマスターがダウンしていれば次のミラーを探すといった流れだ。ミラーリングプロトコル自体は単純な rsync よりは複雑なものだ。というのも、ダウンロードの統計情報を正確に記録したり最低限のセキュリティを確保したりしたかったからである。

同期処理

ミラーリング環境では、中央サーバーとミラーとの間でのデータ転送量を軽減する必要がある。そのため、ミラーでは PyPI の XML-RPC コール `changelog` を使わないといけない。

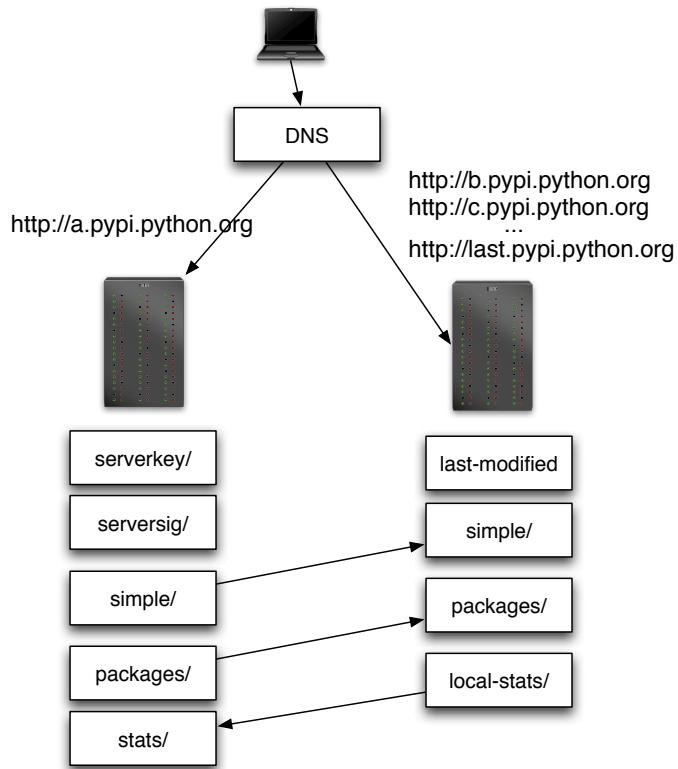



図 14.6: ミラーリング

そして、前回以降で変更があったパッケージだけを再取得するのだ。ひとつのパッケージ P に対してドキュメント`/simple/P/`および`/serversig/P` もコピーする必要がある。

パッケージが中央サーバーから削除されたら、そのパッケージおよびすべての関連ファイルを削除する必要がある。パッケージの内容が変更されたかどうかを検出するために、ファイルの ETag をキャッシュする。そして、`If-None-Match` ヘッダーを使ってリクエストをスキップすることもある。同期が完了したら、ミラーは自身の`/last-modified` を現在日時に設定する。

統計情報の伝搬

どこかのミラーから何かのリリースをダウンロードすると、ミラーリングプロトコルはダウンロードがあったことを PyPI マスターサーバーに送信し、そしてそれが他のミラーにも伝搬する。こうすることで、PyPI を見れば、あるリリースが全部で何回ダウンロードされたのかという情報を全ミラーの合計値で取得できるようになる。

統計情報は日次および週次でCSVファイルにまとめられ、PyPIの中央サーバーの`stats`ディレクトリに置かれている。各ミラーサーバーは、`local-stats`というディレクトリを用意して自分のサーバーに関する統計情報を置かなければいけない。各ファイルに含まれる内容はアーカイブごとのダウンロード回数で、利用したエージェントごとにそれをまとめている。中央サーバーは各ミラーの統計情報を日に一度収集し、それをグローバルな`stats`ディレクトリに取りまとめる。つまり、ミラー側は少なくとも日に一度のペースで`/local-stats`を最新に更新しておく必要がある。

信頼性のミラー

あらゆる分散ミラーリングシステムに共通する問題がある。クライアント側から、ミラーの内容が本物かどうかを調べたくなるということだ。分散ミラーリングシステムで考えられる脅威には、このようなものがある。

- 中央インデックスが攻撃を受ける
- ミラーが改ざんされる
- 中央インデックスとエンドユーザー、あるいはミラーとエンドユーザーの間での中間者攻撃

最初のパターンの攻撃を検出するには、パッケージの作者がPGP鍵でパッケージに署名する必要がある。そうすれば、そのパッケージが本当に作者の作ったものであるかをユーザーが検証できるようになる。ミラーリングプロトコルで対応できるのは二番目の脅威だけであるが、中間者攻撃に対応するための試みもとられている。

中央インデックスはURL `/serverkey`でDSA鍵を提供していつ。これはPEMフォーマットで、`openssl dsa -pubout`[4]で生成したものだ。このURLは決してミラーしてはいけない。クライアントは、PyPIから直接取得した公式の`serverkey`か、PyPIクライアントソフトウェアに付属するそのコピーを利用する。ミラー側は、このキーをダウンロードしておかなければいけない。これを使って、キーの改変を検出する。

各パッケージについて、ミラーされたシグネチャを`/serversig/package`で提供する。これは、対応するURL `/simple/package`のDSAシグネチャをDER形式で表したもので、SHA-1とDSA[5]を使っている。

ミラーを使うクライアントは、次の手順でパッケージを検証しなければいけない。

1. `/simple`をダウンロードし、そのSHA-1ハッシュを計算する。

2. そのハッシュのDSAシグネチャを計算する。

3. 対応する`/serversig`をダウンロードし、先ほど計算した値とバイト単位で比較する。

[4]RFC 3280 SubjectPublicKeyInfo, with the algorithm 1.3.14.3.2.12.
[5]RFC 3279 Dsa-Sig-Value, created by algorithm 1.2.840.10040.4.3.

4. ミラーからダウンロードしたすべてのファイルについて、その MD5 ハッシュを計算して (/simple ページに対して) 検証する。

中央インデックスからダウンロードする場合は検証は不要なので、計算のオーバーヘッドを減らすためにも、そんな場合はむしろ検証すべきではない。

ほぼ一年に一度の頻度で、キーを新しいものと交換する。そのときは、各ミラーが/serversig ページをすべて取り直す必要がある。ミラーを使うクライアントも、新しいキーの信頼できるコピーを取得しないといけない。キーを取得する方法のひとつは、https://pypi.python.org/serverkey からのダウンロードだ。中間者攻撃を検出するために、クライアントは SSL サーバー証明書を検証しなければいけない。この証明書は認証局により署名されている。

14.5 実装の詳細

これまでに説明してきた改良のほとんどの実装は、Distutils2 で行った。setup.py ファイルを使うのはやめて、プロジェクトの設定を完全に setup.cfg に移行した。これは、.ini 風の静的なファイルである。こうすることで、パッケージャがプロジェクトのインストール方法を変更するときにも Python のコードをいじる必要がなくなった。setup.cfg の例を次に示す。

```
[metadata]
name = MPTools
version = 0.1
author = Tarek Ziade
author-email = tarek@mozilla.com
summary = Set of tools to build Mozilla Services apps
description-file = README
home-page = http://bitbucket.org/tarek/pypi2rpm
project-url: Repository, http://hg.mozilla.org/services/server-devtools
classifier = Development Status :: 3 - Alpha
    License :: OSI Approved :: Mozilla Public License 1.1 (MPL 1.1)

[files]
packages =
        mopytools
        mopytools.tests

extra_files =
        setup.py
        README
        build.py
        _build.py

resources =
    etc/mopytools.cfg {confdir}/mopytools
```

Distutils2 は、この設定ファイルを使って次のことを行う。

- メタデータファイル META-1.2 の生成。このファイルは、PyPI への登録などのさまざまな作業に使える。
- 任意のパッケージ管理コマンド (sdist など) の実行。
- Distutils2 ベースのプロジェクトのインストール。

Distutils2 では VERSION も実装している。これは version モジュールを利用したものだ。

INSTALL-DB の実装は Python 3.3 で標準ライブラリ入りする見込みで、pkgutil モジュールに入ることになる。今のところは Distutils2 に入っているが、暫定的なものである。ここで提供する API を使えば、インストールされたものを閲覧して何がインストールされたのかを知ることができる。

これらの API を使って、Distutils2 のこんなイケてる機能を提供している。

- インストーラー/アンインストーラー
- インストールされたプロジェクトに関する依存グラフの表示

14.6 教訓

PEP こそがすべて

Python パッケージングのように大規模で複雑なアーキテクチャを変更するには、細心の注意を要する。PEP プロセスを経て、標準規約を変更することになる。経験上、PEP を変更したり新たに追加したりするには一年はかかる。

これまでにコミュニティが犯した過ちのひとつが、既存の Metadata を拡張して問題を解決しようとしたことだ。Python のアプリケーションをインストールする方法を、PEP を変更することなしに変えてしまったのだ。

言い換えると、使っているツールが何かによって、標準ライブラリ Distutils あるいは Setuptools のどちらを使っているかによって、アプリケーションのインストール方法が違ってくるということだ。新しいツールを使っている一部の人たちは問題を解決できたが、その他大勢の人たちにとっては新たな問題が増えただけだった。たとえば OS のパッケージャーは、複数の Python 標準規格に対応することが必要になった。公式に文書化されている標準と、Setuptools が広めたデファクトスタンダードの二種類である。

しかしその一方で Setuptools には、現実的な規模 (コミュニティ全体) での実験をする機会が与えられた。そのおかげで短期間に革新が進んだし、貴重なフィードバックが得られた。新たに PEP を書きおろすときにも何が動いて何が動かないのか確信を持てたし、他のやり方ではこれほどうまくいかなかっただろう。もし何かのツールが革新を引き起こそうとしているのなら、あわせて PEP も変更すべきだ。

標準ライブラリ入りは死への第一歩

このセクションのタイトルは、Guido van Rossum の声を言い換えたものだ。しかしこれもまた Python の「バッテリー入り」思想の一面であり、私たちの取り組みに大きな影響を及ぼしている。

Distutils は標準ライブラリに組み込まれており、Distutils2 も間もなくその仲間入りする。いったん標準パッケージに入ってしまうと、そこからさらに成長させることはとても難しくなる。もちろん廃止手順もあって、Python のマイナーリリース 2 回ぶんの間に API の変更や廃止もできるようになっている。しかし、いったん API を公開してしまえば、少なくとも数年間は存在し続けるだろう。

そのため、標準ライブラリのパッケージに対してバグ修正以外の変更をしようとすると、Python の世界に混乱を引き起こす可能性がある。大きな変更をするなら、新たなパッケージを作る必要がある。

Distutils でさんざん苦労したおかげで私は学習した。結局、一年がかりの変更をすべて取り消して、新たに Distutils2 を作るはめになった。もしまた標準が劇的に変わるようなことがあれば、どこかで標準ライブラリがリリースされない限りは Distutils3 プロジェクトを始めることになるだろう。

後方互換性

Python でのパッケージングの方法を変更するのはとても長い道のりである。Python の世界には、古いパッケージングツールを使っているプロジェクトも多数存在する。そんな中での変更には、抵抗も多いだろう (本章でとりあげたようなトピックについての合意に達するのにも、結局は数年がかりだった。最初は数カ月で済むものだと思っていたのに)。Python 3 の時点ですべてのプロジェクトが新たな標準規格に移行するには数年かかるだろう。

そのため、私たちの試みはすべて、これまでのツールやインストール方法との後方互換を保つ必要があった。そのおかげで Distutils2 の実装は苦しいものとなった。

たとえば、新しい標準に従っているプロジェクトがまだそれに対応していない別のプロジェクトに依存している場合、「依存関係が不明な状態だ」としてインストールを止めることができなくなってしまう。

たとえば INSTALL-DB の実装には互換性を維持するためのコードが含まれており、元々の Distutils や Pip、Distribute そして Setuptools でインストールされたプロジェクトも閲覧できるようになっている。Distutils2 も同様で、Distutils で作ったプロジェクトもインストールできるようになっている。旧形式のメタデータをその場で変換しているのだ。

14.7　参考文献と謝辞

本章のいくつかのセクションは、私たちが書いた PEP ドキュメントをそのまま引用したものである。原文は `http://python.org` にある。

- PEP 241: Metadata for Python Software Packages 1.0: `http://python.org/peps/pep-0214.html`
- PEP 314: Metadata for Python Software Packages 1.1: `http://python.org/peps/pep-0314.html`
- PEP 345: Metadata for Python Software Packages 1.2: `http://python.org/peps/pep-0345.html`
- PEP 376: Database of Installed Python Distributions: `http://python.org/peps/pep-0376.html`
- PEP 381: Mirroring infrastructure for PyPI: `http://python.org/peps/pep-0381.html`
- PEP 386: Changing the version comparison module in Distutils: `http://python.org/peps/pep-0386.html`

パッケージングにかかわるすべての人たちに感謝する。本章で取り上げたすべての PEP に、君たちの名前が見つかるはずだ。「パッケージングの仲間」のメンバーにも感謝する。また、Alexis Metaireau や Toshio Kuratomi、Holger Krekel、Stefane Fermigier にも感謝する。この文章に関するフィードバックをいただいた。

本章で取り上げたプロジェクトを最後にまとめる。

- `Distutils`: `http://docs.python.org/distutils`
- `Distutils2`: `http://packages.python.org/Distutils2`
- `Distribute`: `http://packages.python.org/distribute`
- `Setuptools`: `http://pypi.python.org/pypi/setuptools`
- `Pip`: `http://pypi.python.org/pypi/pip`
- `Virtualenv`: `http://pypi.python.org/pypi/virtualenv`

第 15 章

Riak と Erlang/OTP

Francesco Cesarini, Andy Gross, and Justin Sheehy

Riak は分散型の耐障害性を備えたオープンソースのデータベースで、Erlang/OTP を使って大規模システムを作るよい例にもなっている。大規模な分散システムを Erlang がサポートしてくれているおかげで、Riak は高可用性や、容量とスループットの両面における線形スケーラビリティといった、データベースではあまり見られない機能を備えている。

Erlang/OTP にはノード間通信、メッセージ・キュー、失敗の検出、クライアント/サーバの抽象化がもともとあるので、Riak のようなシステムを開発するのに理想的なプラットフォームだ。さらに、Erlang でよく使われるパターンは (OTP ビヘイビアと呼ばれる) ライブラリ・モジュールとして実装されている。並行プログラミングを簡単にしたり、開発者がよくある間違いを犯さないようにするため、ビヘイビアには並行実行やエラー処理のための汎用コード・フレームワークが入っている。ビヘイビアはスーパバイザによって監視されていて (スーパバイザ自体もビヘイビアだ)、監視ツリー／＊ supervision trees ＊／の形でまとめられる。監視ツリーはアプリケーションとしてパッケージ化され、Erlang プログラムを構築する際の 1 つのブロックになる。

Riak のように完結した Erlang システムは、互いにやり取りし合う疎結合のアプリケーションが集まってできている。このようなアプリケーションには、開発者自身が書いたものや、Erlang/OTP の標準配布物に入っているもの、他のオープンソースのものがある。アプリケーションを順番にロードするのはブート・スクリプトの役目で、ブート・スクリプトはアプリケーションとそのバージョンのリストから生成される。

個々のシステムを特徴付けるのは、リリース物／＊ which is started ＊／に含まれているアプリケーションたちだ。Erlang の標準配布物では、ブート・ファイルから *Kernel* アプリケーションと *StdLib* (Standard Library) アプリケーションが起動する。インストール環境によっては、*SASL* (Systems Architecture Support Library) アプリケーションも起動する。SASL にはリリースやソフトウェア・アップグレードのためのツールと、ログ機能が入っている。Riak も同様だが、Riak 固有のアプリケーション群と、それらが実行時に依存する *Kernel*、*StdLib*、*SASL* も起動する。完成して実行可能な状態になっている Riak のビルドにはこれらの Erlang/OTP

配布物の標準要素が埋め込まれていて、コマンド・ラインから `riak start` を実行すると順番に起動される。Riak は多数の複雑なアプリケーションで構成されているので、この章ですべてがわかるとは思わないで、Riak のソース・コードに例を取った OTP の紹介と考えてほしい。説明をわかりやすくするため、図や例は省略したり短縮してある。

15.1 Erlang の簡単な紹介

Erlang は並行実行をサポートした関数型プログラミング言語で、コンパイルしたバイト・コードを仮想マシンで実行する。プログラムは関数の呼び出しで構成されるが、プロセス間メッセージ・パッシングや I/O、データベース操作などの副作用を伴なうこともある。Erlang における変数は単一代入だ。つまり、変数にいったん値を与えたら、その値を変更することはできない。次の階乗の例のように、Erlang ではパターン・マッチングを多用する。

```
-module(factorial).
-export([fac/1]).
fac(0) -> 1;
fac(N) when N>0 ->
   Prev = fac(N-1),
   N*Prev.
```

ここでは、最初の節では 0 の階乗を定義し、2 番目の節では正の数の階乗を定義している。各節の本体 (ボディ部) には式を並べるが、節の結果としては最後の式が使われる。負の数に対してこの関数を呼び出すと、どの節にもマッチしないため実行時エラーになる。このような場合についての処理をあえて用意しておかないのは、Erlang で推奨される非防御的プログラミング／＊ non-defensive programming ＊／の 1 つの例だ。

モジュール内での関数の呼び出し方法は特に変ったところはないが、モジュールを越えて関数を呼び出す場合は、`factorial:fac(3)` のようにモジュール名を先頭に付ける。名前が同じだが引数の個数 (*arity* と呼ばれる) が異なる関数を定義することもできる。`factorial` モジュールの export ディレクティブを見るとわかるように、アリティが 1 の `fac` 関数を `fac/1` と表記している。

Erlang はタプル (積型とも呼ばれる) とリストをサポートしている。タプルは`{ok,37}`のように波括弧でくくる。タプルの各要素は位置を指定してアクセスする。レコードというデータ型もある。レコードは決まった個数の要素を持ち、各要素は名前を指定してアクセスしたり操作したりできる。レコードを定義するには`-record(state, {id, msg_list=[]}).` を使う。レコードのインスタンスを作るには `Var = #state{id=1}`という式を使い、その要素を取り出すには `Var#state.id` とする。要素の個数が一定でない場合はリストを使う。リストは `[23,34]` のように角括弧で定義する。`[X|Xs]` という表記はヘッドが `X` でテールが `Xs` の空でないリストにマッチする。小文字で始まる識別子はアトム (それ自身を表わすもの) を示す。たとえば、タプル`{ok,37}`の `ok` はアトムだ。このように、アトムは関数が返す様々な結果を

区別するのに使う。たとえば、関数の結果として ok の他に{error, "Error String"}というのも考えられる。

Erlang システムにおけるプロセスはそれぞれ独立したメモリ空間で並行に動作し、メッセージ・パッシングを使って通信する。プロセスの用途は広く、データベースへのゲートウェイ、プロトコル・スタックのハンドラ、他のプロセスからのトレース・メッセージのログ管理などに使える。プロセスは用途に応じて様々な要求を処理するが、要求の処理方法には似た部分がある。

プロセスは仮想マシン (VM) 内にしか存在しないため、1 つの VM は数百万ものプロセスを同時に実行できる。Riak はこの機能を多いに利用している。たとえば、データベースへの要求の 1 つ 1 つ (読み出し、書き込み、削除) をそれぞれ独立したプロセスとしてモデル化することができる。このような手法は OS レベルのスレッドを用いた実装では非常に困難だろう。

プロセスを識別するにはプロセス識別子 (PID) を使うが、PID は別名を付けて登録することもできる。ただし、PID の登録は寿命の長い静的なプロセスだけにしておくべきだ。プロセスを別名付きで登録しておくと、他のプロセスはそのプロセスの PID を知らなくてもメッセージを送れるようになる。プロセスを生成するには spawn(Module, Function, Arguments) 組み込み関数 (BIF) を使う。BIF とは VM 内に組み込まれている関数で、純粋な Erlang コードでは実現不可能だったり、速度的に問題がある場合に使われる。spawn/3 BIF はパラメータとして Module、Function、リストの Arguments を受け取る。この関数は副作用として新しいプロセスを生成し、そのプロセスの PID を返す。生成されたプロセスは、指定されたモジュールにある指定された関数を、指定された引数を渡して実行する。

プロセス ID が Pid のプロセスにメッセージ Msg を送るには Pid ! Msg を使う。自分自身の PID を知るには、BIF の self を呼び出す。自分自身の PID を他のプロセスに送ることによって、送信先のプロセスは送信元のプロセスとさらに通信できる。たとえば、{ok, N}と{error, Reason}という形のメッセージを受信することを期待しているプロセスがあるとする。このようなメッセージを処理するには、receive 文を使う。

```
receive
   {ok, N} ->
      N+1;
   {error, _} ->
      0
end
```

この文の結果はパターンがマッチした節によって決まる数になる。変数の値がパターン・マッチに不要な場合は、上記の例のようにワイルド・カードとしてアンダースコアを使う。

プロセス間のメッセージ・パッシングは非同期に行なわれ、プロセスが受信するメッセージは到着順にプロセスのメールボックスに置かれる。たとえば上記の receive 式を実行するとしよう。メールボックスの最初の要素が{ok, N}または{error, Reason}ならば、それに対応する結果が返される。メールボックスの最初のメッセージがこの形式でなければ、そのメッセージはメールボックスに留め置かれ、次のメッセージが同様に処理される。どのメッセージもマッチしなければ、マッチするメッセージを受信するまで receive は待機する。

プロセスが終了する理由には2つある。これ以上実行するコードがない場合は、**通常**終了と呼ばれる。実行時エラーが発生した場合は、**異常**／＊ *non-normal* ＊／終了と呼ばれる。プロセスが他のプロセスにリンクしていなければ、プロセスが終了しても他のプロセスには影響を与えない。プロセス同士をリンクするには、`link(Pid)` BIF を使うか、`spawn_link(Module, Function, Arguments)` を使ってプロセスを生成する。プロセスは終了するときに自分のリンク・セットに含まれるプロセス群に EXIT シグナルを送信する。終了理由が異常の場合、プロセスは自分自身を終了し、EXIT シグナルをさらに伝播させる。`process_flag(trap_exit, true)` BIF を呼び出しておくことによって、EXIT シグナルが発生した場合にプロセスは終了せずに EXIT シグナルをメールボックスへの Erlang メッセージとして受信できる。

Riak では、要求を生成する有限状態マシン群が起動したクリティカルでない作業を実行するヘルパー・プロセスの状態を監視するために EXIT シグナルを利用している。／＊長いので分割すべき?＊／このようなヘルパー・プロセスが異常終了した場合、親プロセスは EXIT シグナルを利用して、エラーを無視したりプロセスを起動するといった対処ができる。

15.2　プロセスの骨格

ここまでで、プロセスはその用途に関わらずある共通のパターンに従うことを学んだ。まず、プロセスは生成する必要があり、必要ならばその別名を登録しておくこともできる。新しく生成されたプロセスの最初の動作は、プロセスのループ・データの初期化だ。プロセスの初期化時に `spawn` 組み込み関数に渡された引数の結果をループ・データとして使うことが多い。このループ・データはプロセス状態と呼ばれる変数に格納する。この状態はレコードに格納することが多く、受信/評価関数／＊ receive-evalutate function ＊／に渡される。受信/評価関数が実行するループはメッセージを受信して処理し、状態を更新してその結果を末尾再帰呼び出しに渡す。「停止 (stop)」メッセージを受信したプロセスは後始末をしてから終了する。

プロセスに与えられた役割に関わらず、この処理パターンはよく使われる。このことを念頭に、この処理パターンに従ったプロセス同士で何が違うか見てみよう。

- `spawn` BIF に渡された引数はプロセスごとに異なる。
- プロセスを別名で登録するかどうか、登録するならばどのような名前にするかを決める必要がある。
- プロセス状態を初期化する関数は、プロセスが実行する作業に応じて異なる動作を行なう。
- システムの状態はどの場合でもループ・データで表わされるが、ループ・データの内容はプロセスによって異なる。
- 受信/評価ループの本体部分では、プロセスは異なるメッセージを受信して異なる処理を行なう。
- 最後に、終了時の後始末はプロセスによって異なる。

このように、汎用の動作の骨格はあるが、プロセスの役割に直接関わる特定の動作で補うことになる。この骨格をテンプレートとして使えば、サーバーや有限状態マシン、イベント・ハンドラ、スーパバイザなどとして動作する Erlang プロセスを作ることができる。しかし、これらのパターンは、毎度実装しなおさなくてよいように、ビヘイビアと呼ばれるライブラリ・モジュールになっている。ビヘイビアは OTP ミドルウェアに入っている。

15.3 OTP ビヘイビア

Riak にコミットしている開発者のコア・チームは、十数もの場所に散らばっている。緊密な調整とベースになるテンプレートがなければ、特殊な境界条件や並行実行関連のエラーをまともに扱えないクライアント/サーバ実装がばらばらにできてしまうだろう。クライアントやサーバのクラッシュを扱う方法も一貫しないだろうし、要求に対する応答が単に内部メッセージ・プロトコルに準拠しているだけではなくて、期待したものであることを保証する方法も 1 つに定まらないだろう。

OTP は Erlang のライブラリ群といくつかの基本原理で構成されていて、頑健なシステムを開発するための既製ツールとして利用できる。OTP のパターンやライブラリの多くは “behaviors” として用意されている。

上記のような問題は、よくある並行デザイン・パターンを実装したライブラリ・モジュールである OTP ビヘイビアによって解決できる。プログラマが気にかけなくても、エラーや特別な場合が一貫した方法で処理されるように裏でライブラリ・モジュールが対処してくれる。このため、OTP ビヘイビアは実用／＊ industrial-grade ＊／システムの設計や構築で利用できる、標準化されたブロックの集合となっている。

はじめに

OTP ビヘイビアは、Erlang/OTP 配布物に含まれている `stdlib` アプリケーションのライブラリ・モジュールとして提供されている。プログラマが書くアプリケーションごとのコードは別のモジュールに置かれ、各ビヘイビアごとに標準化されている定義済みコールバック関数を通じて呼び出される。このコールバック・モジュールには、必要な機能を実現するために必要なコードをすべて入れておく。

OTP ビヘイビアには、実際の処理を行なうワーカ・プロセスと、ワーカや他のスーパバイザを監視するためのスーパバイザが含まれている。ワーカ・ビヘイビア (図では円で示す場合が多い) には、サーバ、イベント・ハンドラ、有限状態マシンがある。スーパバイザ (図では四角で示す) は監視ツリーを構成する子プロセス群 (ワーカや他のスーパバイザ) を監視する。

監視ツリーはアプリケーションと呼ばれるビヘイビアにまとめられる。OTP アプリケーションは Erlang システムの構築単位になるだけでなく、再利用可能なコンポーネントをパッケージ化するための方法としても使われる。Riak のような実用システムは、疎結合の、場合

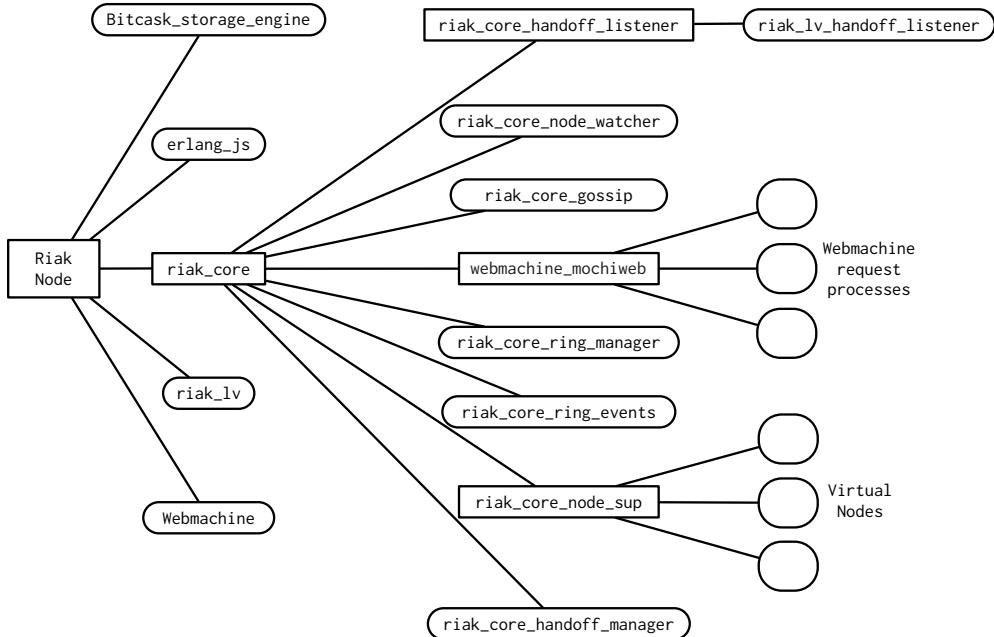



図 15.1: OTP Riak Supervision Tree

によっては分散したアプリケーションの集合で構成される。このようなアプリケーションには、Erlang の標準配布物に入っているものと、Riak 特有の機能を実現するためのものがある。

OTP アプリケーションの例としては、Corba ORB や SNMP (Simple Network Management Protocol) エージェントがある。OTP アプリケーションは、スーパバイザとワーカ・プロセス群をまとめたライブラリ・モジュールの集合を再利用可能なコンポーネントとしてパッケージ化したものだ。以降では、アプリケーションとは OTP アプリケーションのことを指すものとする。

ビヘイビアのモジュールには、そのビヘイビア・タイプに必要な汎用コードがすべて入っている。自分でビヘイビア・モジュールを実装することもできるが、必要になりそうなデザイン・パターンのほとんどは Erlang/OTP の配布物に入っているもので間に合うため、その必要はほとんどないだろう。ビヘイビア・モジュールが用意する汎用の機能には以下のようなものがある:

- プロセスの生成と、場合によっては登録
- 同期または非同期の呼び出しによるクライアント・メッセージの送受信。内部メッセージ・プロトコルの定義も含む
- ループ・データの格納とプロセス・ループの管理
- プロセスの停止

ループ・データとは、ビヘイビアが呼び出しを越えて覚えておく必要があるデータを格納する変数だ。呼び出しの後、更新されたループ・データが返される。更新されたループ・データ (新ループ・データと呼ばれることが多い) は次の呼び出しの引数として渡される。ループ・データはビヘイビアの状態とも呼ばれる。

あるビヘイビアを実現するための汎用サーバ・アプリケーションのコールバック・モジュールに入れる機能には以下のようなものがある:

- プロセスのループ・データの初期化。プロセスを登録する場合はプロセス名の初期化も
- 特定のクライアント要求の処理。同期型の場合はクライアントへの応答の送信
- プロセスへの要求の合間に行なう、プロセス・ループ・データの処理と更新
- 終了時のプロセス・ループ・データの後始末

汎用サーバ

クライアント/サーバ型のビヘイビアを実装する汎用サーバは、標準ライブラリ・アプリケーションの一部である gen_server ビヘイビアで定義されている。この汎用サーバの解説では、riak_core アプリケーションの riak_core_node_watcher.erl モジュールを使う。このモジュールは、Riak クラスタ内のどのサブサービスやノードが利用可能か追跡して報告するサーバだ。モジュールのヘッダとディレクティブは以下のようになっている:

```
-module(riak_core_node_watcher).
-behavior(gen_server).
%% API
-export([start_link/0,service_up/2,service_down/1,node_up/0,node_down/0,services/0,
         services/1,nodes/1,avsn/0]).
%% gen_server callbacks
-export([init/1,handle_call/3,handle_cast/2,handle_info/2,terminate/2, code_change/3]).

-record(state, {status=up, services=[], peers=[], avsn=0, bcast_tref,
                bcast_mod={gen_server, abcast}}).
```

これが汎用サーバであることは-behavior(gen_server) ディレクティブから容易にわかる。このディレクティブは、ールバック関数をすべて適切にエクスポートされているかコンパイラがチェックするために使われる。state レコードはサーバのループ・データで使われる。

サーバの起動

gen_server ビヘイビアでは、spawn BIF と spawn_link BIF を使う代わりに、gen_server:start 関数と gen_server:start_link 関数を使う。spawn と start の主な違いは、呼び出しが同期型かどうかだ。spawn ではなく start を使うことによって、ワーカ・プロセスの PID が初期化されるまで呼び出しが戻らなくなるため、ワーカ・プロセスの起動をより決定的にすることができ、予期しない競合条件による問題を予防できる。start は次のいずれかの形で呼び出す:

```
gen_server:start_link(ServerName, CallbackModule, Arguments, Options)
gen_server:start_link(CallbackModule, Arguments, Options)
```

ServerNameは{local, Name}または{global, Name}という形のタプルで、プロセスを登録する際のプロセス別名を示すローカルまたはグローバルのNameになる。グローバルな名前を使うと、分散Erlangノードのクラスタからサーバを透過的にアクセスできる。プロセスを登録せずにPIDでプロセスを参照する場合は、この引数を持たないstart_link/3またはstart/3を使う。CallbackModuleはコールバック関数が置かれているモジュールの名前、Argumentsはinit/1コールバック関数に渡される有効なErlang項だ。また、Optionsには、メモリ管理関連のfullsweep_afterフラグやheapsizeフラグ、その他トレース用やデバッグ用のフラグのリストを指定できる。

この例では、?MODULEを利用してstart_link/4を呼び出し、コールバック・モジュールと同じ名前でプロセスを登録する。このマクロは、コードをコンパイルする際にプリプロセッサによってモジュール名に展開される。ビヘイビアの名前と、そのビヘイビアを実装しているコールバック・モジュールはいつも同じにしておいたほうがよい。引数は渡さないので、結果的に空のリストを送信することになる。オプション・リストは空のままだ:

```
start_link() ->
    gen_server:start_link({local, ?MODULE}, ?MODULE, [], []).
```

start_link関数とstart関数の明らかな違いは、start_linkは親(たいていはスーパバイザ)にリンクするのに対し、startはリンクしない点だ。自分自身をスーパバイザにリンクするのはOTPビヘイビアの責任になるため、この点は注意が必要だ。start関数はビヘイビアをシェルからテストする際によく使う。入力ミスでシェル・プロセスをクラッシュさせてしまってもビヘイビアには影響がないからだ。start関数やstart_link関数とそれらの変種はすべて{ok, Pid}を返す。

start関数とstart_link関数は、CallbackModuleにあるinit(Arguments)コールバック関数を呼び出すプロセスを生成する。この関数にはArgumentsが渡される。init関数はサーバのLoopDataを初期化し、{ok, LoopData}という形のタプルを返す必要がある。LoopDataにはループ・データの最初のインスタンスを格納する。このループ・データはコールバック関数の間で受け渡される。init関数に渡された引数の一部を取っておきたい場合は、LoopData変数に入れておけばよい。Riakのノード監視サーバのLoopDataは、state型のレコードを渡して呼び出されたschedule_broadcast/1の結果だ。各フィールドにはデフォルト値が設定される:

```
init([]) ->

    %% ノードのアップ/ダウン・イベントを監視する
    net_kernel:monitor_nodes(true),

    %% ノードのステータスを追跡するためのETSテーブルを用意する
    ets:new(?MODULE, [protected, named_table]),
```

```
    {ok, schedule_broadcast(#state{})}.
```

start_link/4 関数を呼び出すのはスーパバイザ・プロセスだが、init/1 コールバックを呼び出すのは別のプロセス、つまり生成されたばかりのプロセスだ。このサーバの役割は Riak の各サブサービスの利用可能状態を監視、記録、ブロードキャストすることなので、初期化処理ではそのようなイベントが通知されるように Erlang ランタイムに依頼し、その情報を記録するためのテーブルを用意する。このデータ構造が存在しないとサーバの呼び出しはすべて失敗してしまうため、この準備は初期化中にやっておかなければならない。init の呼び出しは同期型で、この呼び出しが戻るまでほかの逐次実行プロセスはどれも実行できないので、init 関数での処理は必要最小限に抑えるべきだ。

メッセージの受け渡し

サーバに同期型メッセージを送信する必要があるときは、gen_server:call/2 関数を使う。非同期型呼び出しには gen_server:cast/2 関数を使う。まず、Riak のサービス API の 2 つの関数を例に見てみよう。残りのコードは後で説明する。クライアント・プロセスがこれらの関数を呼び出すと、コールバック・モジュールと同じ名前で登録されているサーバ・プロセスに同期型メッセージが送信される。ただし、サーバに送信されるデータの検証／＊ validating ＊／はクライアント側で行なわれる。クライアントが不正な情報を送ってきた場合、サーバは終了する。

```
service_up(Id, Pid) ->
    gen_server:call(?MODULE, {service_up, Id, Pid}).

service_down(Id) ->
    gen_server:call(?MODULE, {service_down, Id}).
```

gen_server プロセスはメッセージを受信すると handle_call/3 コールバック関数を呼び出す。この関数は、メッセージを送信順に／＊受信順?＊／処理する:

```
handle_call({service_up, Id, Pid}, _From, State) ->
    %% アクティブなサービスの集合をローカルに更新する／＊要確認＊／
    Services = ordsets:add_element(Id, State#state.services),
    S2 = State#state { services = Services },

    %% このサービスの既存の mref をすべて削除する
    delete_service_mref(Id),

    %% このサービスを示す Pid のモニタを用意する
    Mref = erlang:monitor(process, Pid),
    erlang:put(Mref, Id),
    erlang:put(Id, Mref),

    %% ローカルな ETS を更新しブロードキャストする
```

```
    S3 = local_update(S2),
    {reply, ok, update_avsn(S3)};

handle_call({service_down, Id}, _From, State) ->
    %% アクティブなサービスの集合をローカルに更新する
    Services = ordsets:del_element(Id, State#state.services),
    S2 = State#state { services = Services },

    %% このサービスの既存の mref をすべて削除する
    delete_service_mref(Id),

    %% ローカルな ETS を更新しブロードキャストする
    S3 = local_update(S2),
    {reply, ok, update_avsn(S3)};
```

Note the return value of the callback function. The tuple contains the control atom `reply`, telling the `gen_server` generic code that the second element of the tuple (which in both of these cases is the atom ok) is the reply sent back to the client. The third element of the tuple is the new `State`, which, in a new iteration of the server, is passed as the third argument to the `handle_call/3` function; in both cases here it is updated to reflect the new set of available services. The argument `_From` is a tuple containing a unique message reference and the client process identifier. The tuple as a whole is used in library functions that we will not be discussing in this chapter. In the majority of cases, you will not need it.

The `gen_server` library module has a number of mechanisms and safeguards built in that operate behind the scenes. If your client sends a synchronous message to your server and you do not get a response within five seconds, the process executing the `call/2` function is terminated. You can override this by using `gen_server:call(Name, Message, Timeout)` where `Timeout` is a value in milliseconds or the atom `infinity`.

The timeout mechanism was originally put in place for deadlock prevention purposes, ensuring that servers that accidentally call each other are terminated after the default timeout. The crash report would be logged, and hopefully would result in the error being debugged and fixed. Most applications will function appropriately with a timeout of five seconds, but under very heavy loads, you might have to fine-tune the value and possibly even use `infinity`; this choice is application-dependent. All of the critical code in Erlang/OTP uses `infinity`. Various places in Riak use different values for the timeout: `infinity` is common between coupled pieces of the internals, while `Timeout` is set based on a user-passed parameter in cases where the client code talking to Riak has specified that an operation should be allowed to time out.

Other safeguards when using the `gen_server:call/2` function include the case of sending a message to a nonexistent server and the case of a server crashing before sending its reply. In both cases, the calling process will terminate. In raw Erlang, sending a message that is never pattern-matched in a receive clause is a bug that can cause a memory leak. Two different strategies are used in Riak to mitigate this, both of which involve “catchall” matching clauses. In places where

the message might be user-initiated, an unmatched message might be silently discarded. In places where such a message could only come from Riak's internals, it represents a bug and so will be used to trigger an error-alerting internal crash report, restarting the worker process that received it.

Sending asynchronous messages works in a similar way. Messages are sent asynchronously to the generic server and handled in the `handle_cast/2` callback function. The function has to return a tuple of the format `{reply, NewState}`. Asynchronous calls are used when we are not interested in the request of the server and are not worried about producing more messages than the server can consume. In cases where we are not interested in a response but want to wait until the message has been handled before sending the next request, we would use a `gen_server:call/2`, returning the atom `ok` in the reply. Picture a process generating database entries at a faster rate than Riak can consume. By using asynchronous calls, we risk filling up the process mailbox and make the node run out of memory. Riak uses the message-serializing properties of synchronous `gen_server` calls to regulate load, processing the next request only when the previous one has been handled. This approach eliminates the need for more complex throttling code: in addition to enabling concurrency, `gen_server` processes can also be used to introduce serialization points.

Stopping the Server

How do you stop the server? In your `handle_call/3` and `handle_cast/2` callback functions, instead of returning `{reply, Reply, NewState}` or `{noreply, NewState}`, you can return `{stop, Reason, Reply, NewState}` or `{stop, Reason, NewState}`, respectively. Something has to trigger this return value, often a stop message sent to the server. Upon receiving the stop tuple containing the `Reason` and `State`, the generic code executes the `terminate(Reason, State)` callback.

The `terminate` function is the natural place to insert the code needed to clean up the `State` of the server and any other persistent data used by the system. In our example, we send out one last message to our peers so that they know that this node watcher is no longer up and watching. In this example, the variable `State` contains a record with the fields `status` and `peers`:

```
terminate(_Reason, State) ->
    %% Let our peers know that we are shutting down
    broadcast(State#state.peers, State#state { status = down }).
```

Use of the behavior callbacks as library functions and invoking them from other parts of your program is an extremely bad practice. For example, you should never call `riak_core_node_watcher:init(Args)` from another module to retrieve the initial loop data. Such retrievals should be done through a synchronous call to the server. Calls to behavior callback functions should originate only from the behavior library modules as a result of an event occurring in the system, and never directly by the user.

15.4 Other Worker Behaviors

A large number of other worker behaviors can and have been implemented using these same ideas.

Finite State Machines

Finite state machines (FSMs), implemented in the `gen_fsm` behavior module, are a crucial component when implementing protocol stacks in telecom systems (the problem domain Erlang was originally invented for). States are defined as callback functions named after the state that return a tuple containing the next `State` and the updated loop data. You can send events to these states synchronously and asynchronously. The finite state machine callback module should also export the standard callback functions such as `init`, `terminate`, and `handle_info`.

Of course, finite state machines are not telecom specific. In Riak, they are used in the request handlers. When a client issues a request such as `get`, `put`, or `delete`, the process listening to that request will spawn a process implementing the corresponding `gen_fsm` behavior. For instance, the `riak_kv_get_fsm` is responsible for handling a `get` request, retrieving data and sending it out to the client process. The FSM process will pass through various states as it determines which nodes to ask for the data, as it sends out messages to those nodes, and as it receives data, errors, or timeouts in response.

Event Handlers

Event handlers and managers are another behavior implemented in the `gen_event` library module. The idea is to create a centralized point that receives events of a specific kind. Events can be sent synchronously and asynchronously with a predefined set of actions being applied when they are received. Possible responses to events include logging them to file, sending off an alarm in the form of an SMS, or collecting statistics. Each of these actions is defined in a separate callback module with its own loop data, preserved between calls. Handlers can be added, removed, or updated for every specific event manager. So, in practice, for every event manager there could be many callback modules, and different instances of these callback modules could exist in different managers. Event handlers include processes receiving alarms, live trace data, equipment related events or simple logs.

One of the uses for the `gen_event` behavior in Riak is for managing subscriptions to "ring events", i.e., changes to the membership or partition assignment of a Riak cluster. Processes on a Riak node can register a function in an instance of `riak_core_ring_events`, which implements the `gen_event` behavior. Whenever the central process managing the ring for that node changes the membership record for the overall cluster, it fires off an event that causes each of those callback modules to call the registered function. In this fashion, it is easy for various parts of Riak to respond

to changes in one of Riak's most central data structures without having to add complexity to the central management of that structure.

Most common concurrency and communication patterns are handled with the three primary behaviors we've just discussed: `gen_server`, `gen_fsm`, and `gen_event`. However, in large systems, some application-specific patterns emerge over time that warrant the creation of new behaviors. Riak includes one such behavior, `riak_core_vnode`, which formalizes how virtual nodes are implemented. Virtual nodes are the primary storage abstraction in Riak, exposing a uniform interface for key-value storage to the request-driving FSMs. The interface for callback modules is specified using the `behavior_info/1` function, as follows:

```
behavior_info(callbacks) ->
    [{init,1},
     {handle_command,3},
     {handoff_starting,2},
     {handoff_cancelled,1},
     {handoff_finished,2},
     {handle_handoff_command,3},
     {handle_handoff_data,2},
     {encode_handoff_item,2},
     {is_empty,1},
     {terminate,2},
     {delete,1}];
```

The above example shows the `behavior_info/1` function from `riak_core_vnode`. The list of `{CallbackFunction, Arity}` tuples defines the contract that callback modules must follow. Concrete virtual node implementations must export these functions, or the compiler will emit a warning. Implementing your own OTP behaviors is relatively straightforward. Alongside defining your callback functions, using the `proc_lib` and `sys` modules, you need to start them with particular functions, handle system messages and monitor the parent in case it terminates.

15.5 Supervisors

The supervisor behavior's task is to monitor its children and, based on some preconfigured rules, take action when they terminate. Children consist of both supervisors and worker processes. This allows the Riak codebase to focus on the correct case, which enables the supervisor to handle software bugs, corrupt data or system errors in a consistent way across the whole system. In the Erlang world, this non-defensive programming approach is often referred to the "let it crash" strategy. The children that make up the supervision tree can include both supervisors and worker processes. Worker processes are OTP behaviors including the `gen_fsm`, `gen_server`, and `gen_event`. The Riak team, not having to handle borderline error cases, get to work with a smaller code base. This code base, because of its use of behaviors, is smaller to start off with, as it only deals with specific code. Riak has a top-level supervisor like most Erlang applications, and also has sub-supervisors

for groups of processes with related responsibilities. Examples include Riak's virtual nodes, TCP socket listeners, and query-response managers.

Supervisor Callback Functions

To demonstrate how the supervisor behavior is implemented, we will use the riak_core_sup.erl module. The Riak core supervisor is the top level supervisor of the Riak core application. It starts a set of static workers and supervisors, together with a dynamic number of workers handling the HTTP and HTTPS bindings of the node's RESTful API defined in application specific configuration files. In a similar way to gen_servers, all supervisor callback modules must include the -behavior(supervisor). directive. They are started using the start or start_link functions which take the optional ServerName, the CallBackModule, and an Argument which is passed to the init/1 callback function.

Looking at the first few lines of code in the riak_core_sup.erl module, alongside the behavior directive and a macro we will describe later, we notice the start_link/3 function:

```
-module(riak_core_sup).
-behavior(supervisor).
%% API
-export([start_link/0]).
%% Supervisor callbacks
-export([init/1]).
-define(CHILD(I, Type), {I, {I, start_link, []}, permanent, 5000, Type, [I]}).
start_link() ->
    supervisor:start_link({local, ?MODULE}, ?MODULE, []).
```

Starting a supervisor will result in a new process being spawned, and the init/1 callback function being called in the callback module riak_core_sup.erl. The ServerName is a tuple of the format {local, Name} or {global, Name}, where Name is the supervisor's registered name. In our example, both the registered name and the callback module are the atom riak_core_sup, originating form the ?MODULE macro. We pass the empty list as an argument to init/1, treating it as a null value. The init function is the only supervisor callback function. It has to return a tuple with format:

```
{ok,  {SupervisorSpecification, ChildSpecificationList}}
```

where SupervisorSpecification is a 3-tuple {RestartStrategy, AllowedRestarts, MaxSeconds} containing information on how to handle process crashes and restarts. RestartStrategy is one of three configuration parameters determining how the behavior's siblings are affected upon abnormal termination:

- one_for_one: other processes in the supervision tree are not affected.

- rest_for_one: processes started after the terminating process are terminated and restarted.
- one_for_all: all processes are terminated and restarted.

AllowedRestarts states how many times any of the supervisor children may terminate in MaxSeconds before the supervisor terminates itself (and its children). When ones terminates, it sends an EXIT signal to its supervisor which, based on its restart strategy, handles the termination accordingly. The supervisor terminating after reaching the maximum allowed restarts ensures that cyclic restarts and other issues that cannot be resolved at this level are escalated. Chances are that the issue is in a process located in a different sub-tree, allowing the supervisor receiving the escalation to terminate the affected sub-tree and restart it.

Examining the last line of the init/1 callback function in the riak_core_sup.erl module, we notice that this particular supervisor has a one-for-one strategy, meaning that the processes are independent of each other. The supervisor will allow a maximum of ten restarts before restarting itself.

ChildSpecificationList specifies which children the supervisor has to start and monitor, together with information on how to terminate and restart them. It consists of a list of tuples of the following format:

```
{Id, {Module, Function, Arguments}, Restart, Shutdown, Type, ModuleList}
```

Id is a unique identifier for that particular supervisor. Module, Function, and Arguments is an exported function which results in the behavior start_link function being called, returning the tuple of the format {ok, Pid}. The Restart strategy dictates what happens depending on the termination type of the process, which can be:

- transient processes, which are never restarted;
- temporary processes, are restarted only if they terminate abnormally; and
- permanent processes, which are always restarted, regardless of the termination being normal or abnormal.

Shutdown is a value in milliseconds referring to the time the behavior is allowed to execute in the terminate function when terminating as the result of a restart or shutdown. The atom infinity can also be used, but for behaviors other than supervisors, it is highly discouraged. Type is either the atom worker, referring to the generic servers, event handlers and finite state machines, or the atom supervisor. Together with ModuleList, a list of modules implementing the behavior, they are used to control and suspend processes during the runtime software upgrade procedures. Only existing or user implemented behaviors may be part of the child specification list and hence included in a supervision tree.

With this knowledge at hand, we should now be able to formulate a restart strategy defining inter-process dependencies, fault tolerance thresholds and escalation procedures based on a common

architecture. We should also be able to understand what is going on in the init/1 example of the riak_core_sup.erl module. First of all, study the CHILD macro. It creates the child specification for one child, using the callback module name as Id, making it permanent and giving it a shut down time of 5 seconds. Different child types can be workers or supervisors. Have a look at the example, and see what you can make out of it:

```
-define(CHILD(I, Type), {I, {I, start_link, []}, permanent, 5000, Type, [I]}).

init([]) ->
    RiakWebs = case lists:flatten(riak_core_web:bindings(http),
                                  riak_core_web:bindings(https)) of
                   [] ->
                       %% check for old settings, in case app.config
                       %% was not updated
                       riak_core_web:old_binding();
                   Binding ->
                       Binding
               end,

    Children =
                 [?CHILD(riak_core_vnode_sup, supervisor),
                  ?CHILD(riak_core_handoff_manager, worker),
                  ?CHILD(riak_core_handoff_listener, worker),
                  ?CHILD(riak_core_ring_events, worker),
                  ?CHILD(riak_core_ring_manager, worker),
                  ?CHILD(riak_core_node_watcher_events, worker),
                  ?CHILD(riak_core_node_watcher, worker),
                  ?CHILD(riak_core_gossip, worker) |
                  RiakWebs
                 ],
    {ok, {{one_for_one, 10, 10}, Children}}.
```

Most of the Children started by this supervisor are statically defined workers (or in the case of the vnode_sup, a supervisor). The exception is the RiakWebs portion, which is dynamically defined depending on the HTTP portion of Riak's configuration file.

With the exception of library applications, every OTP application, including those in Riak, will have their own supervision tree. In Riak, various top-level applications are running in the Erlang node, such as riak_core for distributed systems algorithms, riak_kv for key/value storage semantics, webmachine for HTTP, and more. We have shown the expanded tree under riak_core to demonstrate the multi-level supervision going on. One of the many benefits of this structure is that a given subsystem can be crashed (due to bug, environmental problem, or intentional action) and only that subtree will in a first instance be terminated.

The supervisor will restart the needed processes and the overall system will not be affected. In practice we have seen this work well for Riak. A user might figure out how to crash a virtual node, but it will just be restarted by riak_core_vnode_sup. If they manage to crash that, the riak_core supervisor will restart it, propagating the termination to the top-level supervisor. This

failure isolation and recovery mechanism allows Riak (and Erlang) developers to straightforwardly build resilient systems.

The value of the supervisory model was shown when one large industrial user created a very abusive environment in order to find out where each of several database systems would fall apart. This environment created random huge bursts of both traffic and failure conditions. They were confused when Riak simply wouldn't stop running, even under the worst such arrangement. Under the covers, of course, they were able to make individual processes or subsystems crash in multiple ways—but the supervisors would clean up and restart things to put the whole system back into working order every time.

Applications

The `application` behavior we previously introduced is used to package Erlang modules and resources into reusable components. In OTP, there are two kinds of applications. The most common form, called normal applications, will start a supervision tree and all of the relevant static workers. Library applications such as the Standard Library, which come as part of the Erlang distribution, contain library modules but do not start a supervision tree. This is not to say that the code may not contain processes or supervision trees. It just means they are started as part of a supervision tree belonging to another application.

An Erlang system will consist of a set of loosely coupled applications. Some are written by the developers, some are available as open source, and others are be part of the Erlang/OTP distribution. The Erlang runtime system and its tools treat all applications equally, regardless of whether they are part of the Erlang distribution or not.

15.6 Replication and Communication in Riak

Riak was designed for extreme reliability and availability at a massive scale, and was inspired by Amazon's Dynamo storage system [DHJ+07]. Dynamo and Riak's architectures combine aspects of both Distributed Hash Tables (DHTs) and traditional databases. Two key techniques that both Riak and Dynamo use are *consistent hashing* for replica placement and a *gossip protocol* for sharing common state.

Consistent hashing requires that all nodes in the system know about each other, and know what partitions each node owns. This assignment data could be maintained in a centrally managed configuration file, but in large configurations, this becomes extremely difficult. Another alternative is to use a central configuration server, but this introduces a single point of failure in the system. Instead, Riak uses a gossip protocol to propagate cluster membership and partition ownership data throughout the system.

Gossip protocols, also called epidemic protocols, work exactly as they sound. When a node in the system wishes to change a piece of shared data, it makes the change to its local copy of the data and gossips the updated data to a random peer. Upon receiving an update, a node merges the received changes with its local state and gossips again to another random peer.

When a Riak cluster is started, all nodes must be configured with the same partition count. The consistent hashing ring is then divided by the partition count and each interval is stored locally as a `{HashRange, Owner}` pair. The first node in a cluster simply claims all the partitions. When a new node joins the cluster, it contacts an existing node for its list of `{HashRange, Owner}` pairs. It then claims (partition count)/(number of nodes) pairs, updating its local state to reflect its new ownership. The updated ownership information is then gossiped to a peer. This updated state then spread throughout the entire cluster using the above algorithm.

By using a gossip protocol, Riak avoids introducing a single point of failure in the form of a centralized configuration server, relieving system operators from having to maintain critical cluster configuration data. Any node can then use the gossiped partition assignment data in the system to route requests. When used together, the gossip protocol and consistent hashing enable Riak to function as a truly decentralized system, which has important consequences for deploying and operating large-scale systems.

15.7 Conclusions and Lessons Learned

Most programmers believe that smaller and simpler codebases are not only easier to maintain, they often have fewer bugs. By using Erlang's basic distribution primitives for communication in a cluster, Riak can start out with a fundamentally sound asynchronous messaging layer and build its own protocols without having to worry about that underlying implementation. As Riak grew into a mature system, some aspects of its networked communication moved away from use of Erlang's built-in distribution (and toward direct manipulation of TCP sockets) while others remained a good fit for the included primitives. By starting out with Erlang's native message passing for everything, the Riak team was able to build out the whole system very quickly. These primitives are clean and clear enough that it was still easy later to replace the few places where they turned out to not be the best fit in production.

Also, due to the nature of Erlang messaging and the lightweight core of the Erlang VM, a user can just as easily run 12 nodes on 1 machine or 12 nodes on 12 machines. This makes development and testing much easier when compared to more heavyweight messaging and clustering mechanisms. This has been especially valuable due to Riak's fundamentally distributed nature. Historically, most distributed systems are very difficult to operate in a "development mode" on a single developer's laptop. As a result, developers often end up testing their code in an environment that is a subset of their full system, with very different behavior. Since a many-node Riak cluster can be trivially run

on a single laptop without excessive resource consumption or tricky configuration, the development process can more easily produce code that is ready for production deployment.

The use of Erlang/OTP supervisors makes Riak much more resilient in the face of subcomponent crashes. Riak takes this further; inspired by such behaviors, a Riak cluster is also able to easily keep functioning even when whole nodes crash and disappear from the system. This can lead to a sometimes-surprising level of resilience. One example of this was when a large enterprise was stress-testing various databases and intentionally crashing them to observe their edge conditions. When they got to Riak, they became confused. Each time they would find a way (through OS-level manipulation, bad IPC, etc) to crash a subsystem of Riak, they would see a very brief dip in performance and then the system returned to normal behavior. This is a direct result of a thoughtful "let it crash" approach. Riak was cleanly restarting each of these subsystems on demand, and the overall system simply continued to function. That experience shows exactly the sort of resilience enabled by Erlang/OTP's approach to building programs.

Acknowledgments

This chapter is based on Francesco Cesarini and Simon Thompson's 2009 lecture notes from the central European Functional Programming School held in Budapest and Komárno. Major contributions were made by Simon Thompson of the University of Kent in Canterbury, UK. A special thank you goes to all of the reviewers, who at different stages in the writing of this chapter provided valuable feedback.

第16章

Selenium WebDriver

Simon Stewart

Selenium はブラウザの自動化ツールで、ウェブアプリケーションのエンドツーエンドテストを書くときによく使われる。ブラウザの自動化ツールが行うのは、その名前から想像できるとおりのことだ。ブラウザを制御して、繰り返し行われるタスクを自動化できる。解決しようとしている問題は単純なものだが、本章で説明するとおり、その裏側ではさまざまなことが起こっている。

Selenium のアーキテクチャについて語る前に、プロジェクト内のさまざまなパーツがどのように関連するのかを説明しておこう。上位レベルから見ると、Selenium は三つのツールを組み合わせたものである。そのうちの一つである Selenium IDE は Firefox の拡張で、これを使うとテストの記録や再生ができる。この「記録/再生」のパラダイムは限定的なものであり、多くのユーザーにとっては物足りないものだ。そこで登場するのが第二のツールである Selenium WebDriver だ。さまざまな言語向けの API を提供しており、より細やかな制御を行うことができるし、標準のソフトウェア開発の流れに組み込むこともできる。最後のツールは Selenium Grid だ。これを使うと、Selenium API で分散環境にあるブラウザのインスタンスを制御でき、テストを並列に実行できるようになる。Selenium プロジェクト内では、これらのツールはそれぞれ “IDE”、“WebDriver” そして “Grid” と呼ばれている。本章で扱うのは Selenium WebDriver のアーキテクチャである。

本章の内容は Selenium 2.0 のベータ版に基づいており、2010 年の後半に執筆された。もしそれより後にこれを読んでいるのなら、世間はさらに進んでいることだろう。ここで取り上げたアーキテクチャに関する選択が実際にどのような形で実装されたのかを見られるかもしれない。もし 2010 年後半よりも前にこれを読んでいるのなら…おめでとう!あなたはついにタイムマシンを手に入れたんだね!頼むから、宝くじの当選番号を教えてくれないかな?

16.1 歴史

ジェイソン・ハギンズが Selenium プロジェクトを立ち上げたのは 2004 年のこと。そのとき彼は ThoughtWorks で社内用の勤怠管理 (T&E: Time and Expenses) システムを開発していた。それは、Javascript を使いまくるシステムだった。Internet Explorer がシェアを支配していた時代だったが、ThoughtWorks ではそれ以外のブラウザも使われており (特に Mozilla 系が多かった)、T&E アプリがそういったブラウザで動かないというバグ報告も受けていた。当時のオープンソースのテストツールといえば、特定のブラウザ (たいてい IE) に絞ったものかブラウザをシミュレートするもの (HttpUnit など) しかなかった。商用のツールのライセンスを購入するというのは、限られた予算の社内システムでは無理な話だった。そのため、そもそも選択肢にすら入っていなかった。

自動化が困難なときには手動でのテストに頼るのが一般的だ。しかし、チームの規模が小さいときやリリース頻度が極端に高いときなどにはこの方式はスケールしない。また、自動化できるはずの手順をいちいちやってもらうように頼むのも、彼らの人間性を浪費しているように思える。はっきり言って、人はくだらない繰り返し作業をするのが苦手だ。機械に比べて能率も悪いし間違いも多い。手動でのテストも選択肢から消えた。

幸いなことに、テスト対象のブラウザはすべて Javascript をサポートしていた。ジェイソンやチームのメンバーが Javascript を使ってテストツールを書こうとしたのも自然なことだった。そのツールを使って、アプリケーションのふるまいを検証しようとしたのだ。既にあった FIT[1] の影響を受け、生の Javascript ではなく表形式の構文を採用した。そのおかげて、プログラミングの経験が少ない人でも HTML ファイルにキーワードを書き込んでテストを書けるようになった。このツールは当初 “Selenium” と呼ばれていたが、後に “Selenium Core” という名前に変わり、Apache 2 ライセンスで 2004 年に公開された。

Selenium の表の書式は、FIT の ActionFixture と似ており、テーブルの各行が三つのカラムに分かれている。最初のカラムには実行するコマンド名を指定し、次のカラムには要素の ID、そして最後のカラムにはオプションの値を指定する。たとえばこれは、“Selenium WebDriver” という文字列をを name が “q” である要素に入力する例だ。

```
type       name=q       Selenium WebDriver
```

Selenium は Javascript だけで書かれているので、初期の設計では Core やテスト群をテスト対象のアプリケーション (AUT) と同じサーバーに置く必要があった。ブラウザのセキュリティポリシーや Javascript のサンドボックスの制限を回避するためである。しかし、現実的にそれが不可能な場合だってある。さらに悪いことに、開発者用の IDE には巨大なコードベースを縦横無尽に渡り歩くための機能が用意されているのに、HTML 用のツールにはそういったものが存在しない。程なくわかったことだが、そんなに大きくないテストスイートの保守ですら面倒できつい作業になった。[2]

[1] http://fit.c2.com

[2] これは FIT でも同じだった。プロジェクトのメンバーの一人であるジェイムズ・ショアが、その弱点について http://jamesshore.com/Blog/The-Problems-With-Acceptance-Testing.html で説明している。

この問題やその他の問題を解決するために、HTTP プロキシが書かれた。これを使い、すべての HTTP リクエストを Selenium で捕捉できるようにしたのだ。このプロキシを使えば、“同一生成元ポリシー” の制約の多くを回避できるようになった。このポリシーは、Javascript からそのページを提供するサーバー以外への呼び出しを許可しないというものである。これを回避できたことで、最初の弱点は何とかしのげるようになった。この設計のおかげて、Selenium のバインディングを複数の言語で書けるようにもなった。必要なのは、単に特定の URL に HTTP リクエストを送信する機能だけである。連結用の書式は Selenium Core の表形式の構文をもとにして作られ、その表形式の構文と併せて後に “Selenese” として知られるようになった。他言語のバインディングはブラウザを遠隔操作するものだったので、このツールは “Selenium Remote Control” あるいは “Selenium RC” と呼ばれることになった。

Selenium の開発が進む一方で、別のブラウザ自動化フレームワークも ThoughtWorks で生まれていた。それが WebDriver で、最初のコードは 2007 年始めに公開された。WebDriver は、あるプロジェクトから派生したツールである。そのプロジェクトでは、エンドツーエンドテストをテストツールから分離しようとしていたのだ。一般的な例にならい、分離する方法として Adapter パターンを使った。WebDriver は、これまでに多数のプロジェクトに適用されてきた結果から生み出されたもので、当初は HtmlUnit に対するラッパーだった。その後、Internet Explorer や Firefox のサポートもすぐに追加された。

WebDriver が公開されたとき、WebDriver と Selenium RC の間には大きな違いがあった。しかしどちらも、ブラウザの自動化のための API を提供するというニッチを狙っているという点では共通していた。ユーザーから見たときの最大の相違点は、Selenium RC は辞書ベースの API ですべてのメソッドを単一のクラスで公開しているのに対して WebDriver はよりオブジェクト指向な API を提供していることだった。さらに、WebDriver がサポートするのが Java だけであるのに対して Selenium RC はさまざまな言語に対応していた。それ以外にも技術的な違いがあった。Selenium Core (RC の元になっているもの) は基本的に Javascript アプリケーションで、ブラウザのセキュリティサンドボックス内で動作する。WebDriver はネイティブにブラウザにバインドすることを試みており、フレームワーク自身の開発に要する労力を犠牲にしてでもブラウザのセキュリティモデルを回避しようとしている。

2009 年 8 月、二つのプロジェクトが合流することが発表された。その結果として登場したのが Selenium WebDriver だ。執筆時点では、WebDriver がサポートしている言語は Java と C#、Python、そして Ruby である。また、Chrome や Firefox、Internet Explorer、Opera、そして Android や iPhone のブラウザに対応している。Selenium WebDriver には姉妹プロジェクトもある。ソースコードリポジトリは別だが本体のプロジェクトと密接に連携して開発が進められており、Perl のバインディングや BlackBerry のブラウザ用の実装を用意している。また、“headless(画面なし)” の WebKit にも対応する。これはテストを継続的インテグレーションサーバーで画面なしで実行しなければならないときに便利だ。元々の Selenium RC の仕組みは今でも保守されており、WebDriver が未対応なブラウザもこれでサポートすることができる。

16.2　ジャーゴンについての余談

残念ながら、Selenium プロジェクトにはジャーゴンが氾濫している。これまでに登場したものをまとめてみよう。

- *Selenium Core* は当初の Selenium の実装の中心となる部分のことで、ブラウザを制御するための Javascript 群である。時には “Selenium” と呼ばれることもあるし、“Core” と呼ばれることもある。
- *Selenium RC* は、Selenium Core 用の言語バインディングを表す。紛らわしいことに、これもまた “Selenium” と呼ばれることもあるし “RC” と呼ばれることもある。現在は Selenium WebDriver に置き換わっており、RC の API は “Selenium 1.x API” と呼ばれる。
- *Selenium WebDriver* は、かつて RC が扱っていたのと同じニッチを埋めるものであり、1.x 系のバインディングを包含している。この言葉は、言語バインディングだけでなく個別のブラウザ制御用コードの実装のことも指す。一般的には単に “WebDriver” と呼ばれており、時には Selenium 2 と呼ばれることもある。そのうち、さらに短縮して “Selenium” と呼ばれるようになるであろうことは疑う余地もない。

賢明な読者のみなさんはお気づきだろうが、“Selenium” という用語があまりにもいろんな場面で使われすぎている。しかし幸いなことに、その「Selenium」が何を表すのかは話の流れで明確になることが多い。

最後に、これから使うであろうフレーズをもうひとつ紹介しておく。“ドライバ” だ。この用語は、WebDriver API の特定の実装を指す名前として使う。たとえば、Firefox ドライバや Internet Explorer ドライバといった使い方をする。

16.3　アーキテクチャについて

個々のピースを見てそのつながり具合を理解する前に、まずはプロジェクト全体のアーキテクチャや開発について知っておくと有用だ。簡潔にまとめると、このようになる。

- コストを抑える。
- ユーザーをエミュレートする。
- ドライバが動作することを実証する…
- …が、それがどのように動作するのかを知る必要はないようにする。
- バス係数を上げる。
- Javascript の実装を尊重する。
- すべてのメソッドコールは RPC である。
- 我々は、オープンソースプロジェクトである。

コストを抑える

〇〇プラットフォーム上の■■ブラウザをサポートするというのは本質的にコストのかかる提案である。最初の開発の面でも、その後の保守の面でも。その品質を高く保ちつつその他の原則を破らずに済む手段がもしあれば、それこそが我々の目指す道となる。我々が可能な限り Javascript を採用しているのも、それが理由だ。後ほど詳しく説明する。

ユーザーをエミュレートする

WebDriver は、ユーザーがウェブアプリケーションとやりとりする内容をそのまま正確にシミュレートするように作られている。ユーザーの入力をシミュレートする一般的な手段は、Javascript を使って一連のイベントを合成することだ。アプリケーション側からは、ユーザーがその操作をしたのと同じように見える。このような "イベント合成" 方式は、面倒なことだらけである。ブラウザによって、また場合によっては同じブラウザでもバージョンによって、発火するイベントが微妙に異なったり微妙に違う値をとったりする。さらに問題を複雑にしているのが、多くのブラウザがこのような手段でのフォーム要素の操作を許可していないという事実だ。たとえばファイルアップロード要素などは、セキュリティの観点から "イベント合成" ができないようになっている。

WebDriver では、可能な場合は Javascript を使わず OS レベルでイベントを発火させるという手法をとる。このような "ネイティブイベント" はブラウザが生成するものではないので、イベント合成方式に立ちはだかったセキュリティの制約を回避できる。そして、OS の機能を使っているので、特定のプラットフォーム上のあるブラウザで動作するようになれば、そのコードを再利用して同じプラットフォーム上の別のブラウザに対応させるのも容易である。不幸にも、この手法が使える環境は限られている。WebDriver とブラウザを密接にバインドでき、かつウィンドウにフォーカスを合わせずにブラウザへネイティブイベントを送る方法を開発チームが見つけた場合 (Selenium のテストは実行に時間がかかるので、テストの実行中にはマシン上で他のタスクを実行できるようにしておきたい) にしか使えない。執筆時点では、この条件を満たしてネイティブイベントが使える環境は Linux と Windows である。Mac OS X では使えない。

WebDriver がユーザーの入力をどうやってエミュレートするかにかかわらず、我々は可能な限りユーザーのふるまいを再現できるよう努力している。これは RC とは対照的で、RC が提供する API は実際のユーザーの作業よりずっと低レベルな操作をするものである。

ドライバが動作することを実証する

ある意味理想主義的かもしれないが、動作しないコードなどまったく無意味である。Selenium プロジェクトでドライバが正しく動作することを実証するために、自動テストのテストケースを豊富にそろえている。その多くは "インテグレーションテスト" であり、コードがコンパ

イルされていることとブラウザを使ってウェブサーバーとやりとりすることを前提としている。しかし、書ける場面では “ユニットテスト” も書いている。これはインテグレーションテストとは異なり、完全に再コンパイルしなくても実行できる。執筆時点では、約 500 のインテグレーションと約 250 のユニットテストがあってそれぞれすべてのブラウザで実行できる。バグ修正や機能追加のときには新たなテストを追加するし、我々としてはもっとユニットテストを書くように力を注いでいる。

すべてのテストがすべてのブラウザで動作するというわけではない。いくつかのブラウザでは対応していない特定の機能を確認するテストもあるし、ブラウザによって処理方法が変わる機能のテストもある。たとえば HTML5 の機能のテストがこれにあたる。HTML5 の中には、すべてのブラウザが対応しているわけではない機能もある。それにもかかわらず、主要なデスクトップブラウザには、それぞれ大量のテストのサブセットがある。ご存じの通り、500 を超えるテストを複数のプラットフォームでブラウザごとに実行するのは大変な作業だ。我々が常に闘い続けている課題でもある。

すべての動作原理を理解する必要はない

我々が使っているすべての言語や技術について熟練しているという人はほとんどいない。したがって、我々のアーキテクチャは各開発者が自分の得意分野だけに注力できるようにすべきだ。満足に扱えない不得手な部分をいじらなくても、やりたいことができるようにする。

バス係数を上げる

ソフトウェア開発の世界には “バス係数 (bus factor)” と呼ばれる概念がある (まじめなものではない)。この係数は、プロジェクトのコア開発者の数を表す。仮にプロジェクトに何らかの災難 (バスに追突されるなど) があったときに、その人が欠けてしまえばプロジェクトが続行不能になるような人のことである。ブラウザの自動化のように複雑な作業は、特にこのようになりやすい。そこで、我々のアーキテクチャに関する決断は、この係数を可能な限り上げる方向に進めている。

Javascript の実装を尊重する

WebDriver は、他に何も手段がない場合は最終的にピュア Javascript を使ってブラウザを操作する。つまり、我々が用意する API は Javascript の実装にあわせたものでなければいけないということだ。ここで具体例を示す。HTML5 で導入された LocalStorage は、構造化されたデータをクライアント側に保存するための API である。この API のブラウザでの実装には、通常は SQLite が使われている。我々としての自然な実装は、ベースとなっているデータストアへのデータベース接続を (JDBC などで) 提供することだろう。しかし、最終的に我々が用

意した API は背後にある Javascript の実装を模したものだった。なぜなら、典型的なデータベースアクセス用 API を参考にすると Javascript の実装とは食い違ってしまうからだ。

すべてのメソッドコールは RPC である

WebDriver が制御するブラウザは、別のプロセスで動いている。見落としがちなことだが、これはつまり、API からのすべての呼び出しは RPC コールになるということだ。したがって、フレームワークのパフォーマンスはネットワークのレイテンシに大きな影響を受ける。通常の操作では、特に気になるほどの問題は発生しない (たいていの OS は、localhost へのルーティングを最適化している)。しかしブラウザとテストコードの間のネットワークレイテンシが増加すると、API の設計側も API の利用者側も使いづらくなる。

このせいで、API の設計に関してちょっとした対立が発生する。大きな API で荒い関数群を用意すると、複数のコールをひとまとめにしてレイテンシを下げることができる。しかしこれは、API を表現力があって使いやすいものにすることとのトレードオフとなる。たとえば、ある要素がエンドユーザーに見えるかどうかを調べるには、いくつかのチェックが必要となる。CSS のさまざまなプロパティを (親要素のプロパティも含めて) 考慮するだけでなく、おそらく要素の大きさも調べなければならないだろう。ミニマルな API なら、これらのチェックをそれぞれ個別に行うことになるだろう。WebDriver では、これらのチェックを単一のメソッド `isDisplayed` にまとめた。

結論: オープンソースです

アーキテクチャの観点からは少しずれるが、Selenium はオープンソースプロジェクトである。ここまであげてきたポイントをまとめると、我々が望むのは新しい開発者がプロジェクトに参加しやすいようにするということだ。参加するために必要な知識をできるだけ抑え、知っていてほしい言語の数も最小にして、さらに自動テストを活用して何も壊さないことを確かめる。これらはすべて、新規参入を促すのが狙いである。

最初は、このプロジェクトは複数のモジュール群に分かれていた。特定のブラウザ用のモジュールがいくつかあって、さらに全体に共通するコードやユーティリティコードを含むモジュールがあったのだ。各バインディングのソースツリーもこれらのモジュールの配下にあった。この方式は Java や C#使いの人たちにとっては便利だったが、Rubyist や Pythonista にとってはやりにくいものだった。これはそのまま、各言語の貢献者数の違いとなって現れた。Python や Ruby のバインディングを開発できる人 (あるいは開発に興味を持つ人) がほんの少ししかいなかったのだ。これを解決するために、2010 年の 10 月から 11 月にかけてソースコードの構成を見直し、Ruby や Python のコードはトップレベルに言語ごとのディレクトリを置いてそこで管理するようにした。Ruby や Python の世界にいるオープンソース開発者には、このほうが見慣れた構造だった。そして、それからすぐに、これらの言語のコミュニティからの貢献が目に見えて増加した。

16.4　複雑性への対応

ソフトウェアは、こぶだらけででこぼこな構造である。その原因は複雑性であり、API を設計する側の立場で考えると、その複雑性をどこに押し込めるかを判断しなければならない。究極の選択として、複雑性を可能な限りまんべんなく広げるという方法がある。これは、API の利用者に複雑性をすべて押しつけてしまうというものである。その対極にあるのが、API 側で可能な限り複雑性を囲い込み、一か所に隔離してしまうという方法である。囲い込んだ場所は暗黒地帯となり、そこに手を入れるのはとても恐ろしい作業になるだろう。しかしその見返りとして、API のユーザーは実装に深入りせずに済むようになる。つまり、ユーザーが複雑性に立ち向かうためのコストを設計者側で前払いしたことになる。

WebDriver の開発者は、複雑性を見つけたらそれをできるだけ特定のいくつかの場所に隔離するよう心がけている。そのまま広めることはしない。その理由のひとつはユーザー層である。彼らはバグや問題点を見つけることにかけては並はずれた力を持っている。それは我々のバグリストを見れば明らかだ。しかし、彼らの多くは開発者ではないので、あまり複雑な API はうまく使いこなせない。我々が求める API は、利用者を正しい方向に導けるようなものだ。たとえば、オリジナルの Selenium API のこれらのメソッドについて考えてみよう。これらはどれも、input 要素に値を設定するために使うものだ。

- type
- typeKeys
- typeKeysNative
- keydown
- keypress
- keyup
- keydownNative
- keypressNative
- keyupNative
- attachFile

同じことを実現するための WebDriver の API は、これだ。

- sendKeys

先述のとおり、これは RC と WebDriver の思想の大きな違いのひとつを表している。WebDriver が目指すのがユーザーをエミュレートすることであるのに対して、RC が提供する API はより低レベルのものであり、ユーザーの視点では見つけづらかったり到達不能だったりするものである。`typeKeys` と `typeKeysNative` の違いは、前者が常に合成イベントを使うのに対して後者は AWT の Robot を使ったキータイプを試みるという点である。残念なことに、AWT の Robot がキープレスイベントを送信する先はフォーカスがあたっているウィンドウになる。これは必ずしもブラウザであるとは限らない。WebDriver は、それとは対照的に、ウィンドウハンドルに対して直接イベントを送信する。そのため、ブラウザのウィンドウにフォーカスがあたっている必要はない。

WebDriverの設計

開発チームでは、WebDriverのAPIが“オブジェクトベース”であるよう心がけている。インターフェイスを明確に定義して単一のロールあるいは責務だけを受け持たせようとする。しかし、考えうるすべてのHTMLタグを個別にモデル化するわけではなく、我々が用意したのは一つの`WebElement`インターフェイスだけである。この方針で進めると、自動補完機能を持つIDEを使う開発者に対して次に進むべき道を示せる。コーディングは、(たとえばJavaの場合) このような具合になる。

```
WebDriver driver = new FirefoxDriver();
driver.<ここでスペースキーを打つ>
```

このときに現れるのは、比較的少なめな13種類のメソッドである。その中から適切なものを選ぶことになる。

```
driver.findElement(<ここでスペースキーを打つ>)
```

たいていのIDEは、ここで引数の型に関するヒントを表示する。この場合は“By”である。“By”オブジェクトには定義済みのファクトリーメソッドが数多く用意されており、By自身のスタティックメソッドとして宣言されている。ユーザーは、あっという間にこのような状態までたどりつけるだろう。

```
driver.findElement(By.id("some_id"));
```

`UnsupportedOperationExceptions`やその仲間たちはとても不愉快なものだったが、その機能を必要とする一部のユーザーに対して何らかの手段で機能を公開する必要があった。大多数の他のユーザー向けの他のAPIには影響を及ぼさない形式で。そのための手段として、WebDriverではロールベースのインターフェイスを活用した。たとえば`JavascriptExecutor`インターフェイスは、任意のJavascriptコード片を現在のページのコンテキストで実行する機能を提供する。WebDriverのインスタンスをこのインターフェイスにあわせてキャストすれば、このインターフェイスの持つメソッドを使える。

組み合わせの激増への対処

ちょっと考えればすぐにわかることだが、WebDriverがさまざまなブランザや言語に対応しているということは、よっぽど注意しないと保守のコストが激増してしまうということである。X種類のブラウザとY種類の言語に対応しようとすると、最悪X×Y種類の実装を保守するはめになる。

WebDriverでサポートする言語を減らすのもひとつの方法ではあるが、その道をゆくつもりはなかった。理由はふたつある。まず、ある言語から別の言語への切り替えには認識を切り替えるコストがかかる。つまり、フレームワークのユーザーにとっては、開発時に使って

ロールベースのインターフェイス

単純な Shop クラスを考えてみよう。この店では、毎日のように在庫の補充が必要となる。そこで、Stockist と協調して新たな在庫を配達する。また、従業員への給与の支払いや税金の支払いが毎月発生する。議論の都合上、これらの処理には Accountant を使うものとしよう。これをモデリングした一例を次に示す。

```
public interface Shop {
    void addStock(StockItem item, int quantity);
    Money getSalesTotal(Date startDate, Date endDate);
}
```

Shop と Accountant そして Stockist の間のインターフェイスを定義するときにどのあたりに境界線を引くか。この問いにはふたつの選択肢がある。理論上の線を図 16.1 のように引ける。

これはつまり、Accountant や Stockist がそれぞれのメソッドの引数として Shop を受け付けることを意味する。しかし、この方式には問題もある。Accountant がほんとうに在庫棚を使いたがっているとは思えないし、Stockist にとっては Shop が追加したさまざまな商品の値上げ情報など知らせる必要もない。そこで、もう少しましな線の引き方を図 16.2 に示す。

この場合、Shop はふたつのインターフェイスを実装しなければならない。しかし、これらのインターフェイスは、Shop が Accountant と Stockist に対して満たすべきロールを明確に定義している。これが、ロールベースのインターフェイスである。

```
public interface HasBalance {
    Money getSalesTotal(Date startDate, Date endDate);
}

public interface Stockable {
    void addStock(StockItem item, int quantity);
}

public interface Shop extends HasBalance, Stockable {
}
```

いるのと同じ言語でテストを書ければ好都合になる。次に、ひとつのプロジェクトに複数の言語を混在させるとチーム内で不満が出る可能性がある。場合によっては、会社のコーディング規約などで特定の技術だけを使うよう強制されているかもしれない (しかし、ありがたいことに、最近は二番目の理由はあまり聞かなくなってきた)。したがって、サポートする言語を減らすという選択肢は取れなかった。

　サポートするブラウザを減らすというのも、あり得ない話だ。かつて WebDriver で Firefox

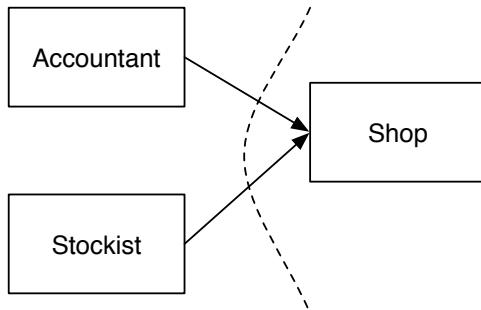



図 16.1: Accountant と Stockist が Shop に依存する

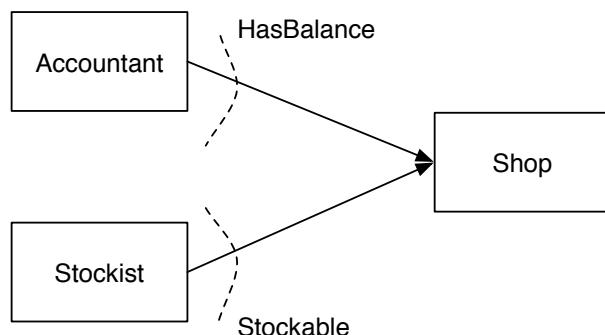



図 16.2: Shop は HasBalance と Stockable を実装する

2 のサポートを打ち切ったときには大騒ぎになった。当時のブラウザ市場での Firefox 2 のシェアは既に 1%を割っていたというのに。

残された道はひとつ。各言語のバインディングから、すべてのブラウザを同じように扱えるようにするという道だ。統一形式のインターフェイスを用意し、さまざまな言語から容易にアクセスできるようにする。さらに考えたのは、言語バインディングそのものもできる限り書きやすくするということだった。そのためには、言語バインディングをできるだけスリムにしておくことを考えた。これを実現するため、できる限りのロジックをベースとなるドライバ側に押し込めた。ドライバに組み込めなかった機能はすべての言語のバインディング側で実装する必要があり、大変な作業となってしまう。

たとえば IE ドライバでは、IE を探して起動する責務をドライバ本体のロジックに組み込めた。その結果としてドライバ側のコードの行数はおそろしいほどに膨れ上がったが、新しいインスタンスを作るための言語バインディングはドライバのメソッドをひとつ呼び出すだけで済むことになった。ちなみに、Firefox ドライバの場合はこの変更に失敗した。Java の世界だけでも、三つの巨大なクラスで Firefox の設定や起動を処理していた。行数にして 1300 行ほどである。Java サーバーを立てずに FirefoxDriver をサポートしようとすると、すべての

言語のバインディングでこれらのクラスが重複することになる。つまり、同じようなコードを大量に保守しなければならないということだ。

WebDriver の設計の問題点

この方式で機能を公開したことによるマイナスは、あるインターフェイスの存在に気づくまでは WebDriver がそんな機能を持っていることに気づけないということである。これは、API の探索性を損ねてしまう。実際、WebDriver が登場したばかりのころは、利用者にそのインターフェイスの存在に気づかせるために多くの時間を費やしていた。最近ではドキュメントにより力を注ぐようになり、API も幅広く使われるようになったので、ユーザーにとっては必要な情報を見つけやすくなってきた。

自分たちの APIって本当に貧弱だなあと感じる箇所をひとつ紹介しよう。`RenderedWebElement` というインターフェイスがあるのだが、これはいろんなメソッドがごちゃ混ぜになったものである。たとえばレンダリング後の要素の状態を取得したり (`isDisplayed` や `getSize` そして `getLocation`)、その要素に対して何らかの操作をしたり (`hover` やドラッグ&ドロップなど)、CSS のプロパティの値を取得したりといったことができる。そもそもこのインターフェイスが作られた理由は HtmlUnit ドライバが必要な情報を公開していなかったことだが、Firefox や IE のドライバは情報を公開していた。最初は要素の状態を取得するメソッド群しか提供していなかったが、どんどん他のメソッドが増えていった。メソッドを増やす前に、API をどのように育てていくかを熟考すべきだったと後悔している。このインターフェイスは今や広く使われているため、どのように対処するかが難しい。見苦しい API ではあるけれども広く使われているのだからそのまま維持し続けるのか、いっそのこと削除してしまうのか。私は “割れ窓” を放置しておきたくなかった。そこで、Selenium 2.0 のリリースまでにこれを何とかすることが重要となった。その結果どうなったか。みなさんがこれを読むころには `RenderedWebElement` は削除されているはずだ。

実装側の観点で考えると、ブラウザと密に結合してしまう設計にも問題がある。たとえそれが不可避なものであったとしてもである。新しいブラウザに対応するのは大変な作業になるし、うまく動かすにはたいてい試行錯誤が必要となるだろう。具体例を挙げると、Chrome ドライバは 4 回ほどゼロから書き直したし、IE ドライバも 3 回は大幅に書き直している。ブラウザと密結合する利点は、ブラウザをより細かく制御できるという点である。

16.5 レイヤーと Javascript

ブラウザの自動化ツールは、基本的にこれらの三つの部分から構成されている。

- DOM への問い合わせ手段。
- Javascript を実行する仕組み。
- ユーザーの入力をエミュレートする何らかの手段。

このセクションで扱うのは最初の項目、つまり DOM への問い合わせの仕組みである。ブラウザの世界の共通語は Javascript であり、これを使って DOM の問い合わせができれば理想的であろう。この選択は一見明らかなようだが、Javascript を採用するにはいくつかの問題や競合する要件があり、バランスをとる必要がある。

多くの大規模プロジェクトと同様、Selenium のライブラリ群は階層構造になっている。最下層にあるのが Google の Closure Library で、これはプリミティブやモジュール化機構を提供する。これを使うと、ソースファイルを集中的に保持しつつサイズを小さくできる。その上に乗るのがユーティリティライブラリで、このライブラリではさまざまな関数を提供する。属性の値を取得したりある要素がユーザーに見えるかどうかを調べたりといった単純なタスクをこなす関数もあれば、同期イベントを使ってクリック操作をシミュレートするなどの複雑な関数もある。プロジェクト全体から見たときに、このライブラリはブラウザ自動化の最小単位の機能を提供するものに見える。そのため、このライブラリは Browser Automation Atoms あるいは atoms と呼ばれる。最後に、その上に乗るのがアダプタ層である。これが、WebDriver およびコアとの API の役割をする。

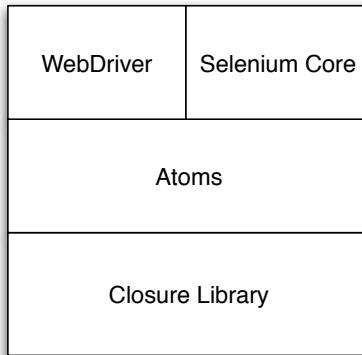



図 16.3: Selenium Javascript ライブラリの階層

Closure Library が選ばれたのにはいくつか理由がある。最大の理由は、Closure Compiler がそのライブラリの使っているモジュール化技術を理解するということだ。Closure Compiler は、出力言語として Javascript を対象としたコンパイラである。“コンパイル” とひとことで言ってもその幅は広い。単に入力ファイルを依存関係の順で並べ替えて連結・整形するだけのこともあれば、高度な最適化や未使用コードの削除まで含めることもある。Closure Library には、それ以外にもまぎれもない利点があった。Javascript のコードを開発するチームのメンバーの中に、Closure Library にとても詳しい人が何人かいたのだ。

この “核となる” ライブラリは、プロジェクト全般で DOM を扱う必要のある場面に幅広く用いられている。RC や主に Javascript で書かれたドライバの場合は、このライブラリを直接使う。ひとつのスクリプトにまとめてしまうことが多い。Java で書かれたドライバの場合は、WebDriver アダプタ層の個々の関数を完全に最適化した状態でコンパイルし、生成され

たJavascriptをJARにリソースとして組み込む。C系の言語で書かれたドライバ、たとえばiPhone用やIE用のドライバの場合は、個々の関数を最適化してコンパイルするだけではなく、生成された出力をヘッダで定義する定数に変換する。そしてそれを、ドライバの標準的なJavascript実行機能を使って必要に応じて実行する。奇妙なことに思えるかもしれないが、Javascriptを基盤のドライバに押し込むこともできる。そうすれば、複数の場所で生のソースを公開する必要がなくなる。

atomsは広範囲で使われているので、さまざまなブラウザで一貫したふるまいを保証できる。また、ライブラリがJavascriptで書かれていて実行するのに権限の昇格が不要なので、開発サイクルを手軽にすばやく回せるようになる。Closure Libraryは動的に依存を読み込めるので、Seleniumの開発者はただ単にテストを書いてそれをブラウザに読ませるだけでよい。コードを修正したら、必要に応じて再読み込みさせる。あるブラウザでテストが通れば、あとはそれを他のブラウザでも読ませてテストが通ることを確認するだけである。Closure Libraryがうまい具合にブラウザ間の差異を取り去ってくれているので通常はこれで十分だが、サポート対象の全ブラウザ用のテストスイートを実行する継続的ビルド環境もあることを知ればさらに安心できるだろう。

もともと、CoreとWebDriverで同じようなことをしているコードは多かった。同じ機能を微妙に異なる方法で実現していたのだ。atomsを使った開発を始める際に、我々はコードを念入りに調べて"最適なもの"を残そうとした。結局のところ、どちらのプロジェクトについても、幅広く使われていたこともあってコードは非常に堅牢だった。すべての投げ捨ててゼロから書き直すのは無駄だし馬鹿げていた。個々のatomをピックアップして、それを使うであろう場面を調べてatomを使うように切り替えた。たとえばFirefoxドライバの`getAttribute`メソッドは、もともと50行ほどあったのが空行込みで6行にまで縮まった。

```
FirefoxDriver.prototype.getElementAttribute =
  function(respond, parameters) {
  var element = Utils.getElementAt(parameters.id,
                                   respond.session.getDocument());
  var attributeName = parameters.name;

  respond.value = webdriver.element.getAttribute(element, attributeName);
  respond.send();
};
```

最後から2行目の`respond.value`に代入しているところで、WebDriver用のatomを使っている。

atomsは、このプロジェクトのアーキテクチャに関する主題のいくつかを現実的に実証したものである。当然、APIの実装はJavascriptの実装に従うという要件を強要する。さらによい点は、同じライブラリをコードベース全体で共有しているというところだ。複数のプラットフォームにまたがるバグが検出されて修正することになっても、一か所でバグを修正するだけでよくなる。変更に対するコストは下がり、安定性や効率性も向上した。atomsを使えば、プロジェクトのバス係数をより好ましい方向に進められる。通常のJavascriptのユニット

テストツールをつかえば修正がうまく機能するかどうかを検証できるので、プロジェクトへの新規参入障壁はかなり下がるだろう。参加する前に各ドライバの実装の詳細を知っておく必要がなくなる。

atoms を使うメリットはそれ以外にもある。既存の RC の実装をエミュレートした (しかし裏側には WebDriver がいる) レイヤーをうまく使えば、きちんと管理された方法で新しい WebDriver の API に移行させることができる。Selenium Core は atom 化されているので、その各関数を個別にコンパイルできるようになった。そのおかげで、このエミュレートレイヤーを書く作業は容易になり、より正確に書けるようになった。

もちろん、このアプローチには弱点もあることは言うまでもない。中でも最も重大なのが、Javascript をコンパイルして C の定数にしてしまうという奇妙な作業だ。これは、C のコードでプロジェクトに貢献したいと考えている人の気持ちを挫折させてしまうに十分だ。また、すべてのブラウザの全バージョンを持っていてそのすべてで全テストを実行できるような開発者などめったにいない。誰かが不注意で予期せぬ場所にバグを作りこんでしまうこともあり得るし、もし継続的ビルドがうまく機能していなければ、そんなバグが見つかるまでにはある程度の時間を要してしまうだろう。

atoms はブラウザごとの返り値を正規化するので、予期せぬ値が返ってくることもあり得る。たとえば、このような HTML を考えよう。

```
<input name="example" checked>
```

checked 属性の value が何になるかはブラウザに依存する。atoms はこれを正規化し、その他 HTML5 仕様で定義されているすべての Boolean 属性の値を “true” か “false” に揃える。この atom をはじめてコードベースに投入したときに気付いたのは、返り値に関してブラウザ依存の値を前提としているコードがいかに多いかということだった。この値は今は一貫しているが、それまでにはコミュニティに対して「何が起こったのか」「なぜそうなったのか」を説明するための期間を要した。

16.6 リモートドライバ、中でも特に Firefox ドライバについて

リモート WebDriver は、当初から RPC の仕組みを尊重していた。それは徐々に成長して WebDriver の主要な仕組みのひとつとなり、保守コストの軽減に役立った。統一インターフェイスを提供することで、各言語のバインディングがそれを使えるようにしたのだ。ロジックのほとんどを言語バインディングからドライバ側に移してはいるが、もしドライバ側から独自のプロトコルでの通信が必要になったときは、それに対応するコードがすべての言語バインディングにまたがって用意されている。

リモート WebDriver プロトコルは、別プロセスで動いているブラウザのインスタンスとの通信が必要になったときに使うものである。このプロトコルの設計にあたっては、さまざまな内容を検討した。ほとんどは技術的なものだが、オープンソースプロジェクトであることを鑑みて、ソーシャルな面も検討材料のひとつになった。

あらゆる RPC 機構は、二つの部分に分かれる。トランスポートとエンコーディングだ。リモート WebDriver プロトコルをどのように実装したところで、クライアントとして想定するすべての言語に対してこれら両方のサポートが必要になることはわかっていた。設計の最初のイテレーションでは、Firefox ドライバについて検討した。

Mozilla は (つまり Firefox も) 常に、マルチプラットフォームアプリケーションとして開発されてきた。その開発を促進するために、Mozilla は Microsoft の COM に似たフレームワークを作った。このフレームワークを使うと、コンポーネントの組み立てや連携が XPCOM(クロスプラットフォーム COM) という仕組みで可能となる。XPCOM のインターフェイスは IDL で宣言されており、C や Javascript だけでなくその他の言語のバインディングも用意されている。XPCOM は Firefox 自体を作るためにも使われており、さらに Javascript バインディングを持っているので、XPCOM オブジェクトを Firefox 拡張で使うことができる。

Win32 COM には、リモートからアクセスするためのインターフェイスが用意されている。XPCOM にも同様の機能を追加する計画があった。実際、ダーリン・フィッシャーが XPCOM ServerSocket の実装を追加してこれを実現しようとしていた。結局この D-XPCOM 計画が日の目を見ることはなかったが、その基盤の痕跡は今でも残っている。我々はこの機能を使い、基本的なサーバー機能を Firefox の拡張機能として実装した。そしてそこに、Firefox の制御のためのすべてのロジックを閉じ込めた。利用したプロトコルは、テキストベースで行指向なもので、すべての文字列を UTF-2 でエンコードしていた。個々のリクエストやレスポンスは数値から始まる。この数値は、リクエストやレスポンスに達するまでに改行文字がいくつ現れるかを指している。最も重要だったのは、このスキームは Javascript で SeaMonkey(当時の Firefox が採用していた Javascript エンジン) として実装しやすかったということだ。これは、Javascript の文字列を内部的に符号なし 16 ビット整数値で格納している。

独自の符号化プロトコルを生のソケットにのせてやりとりするのは、暇つぶしとしてはいいだろう。しかし、いくつかの問題点もある。まず、自前のプロトコルに対応したライブラリは広く出回っていないので、サポートしたいすべての言語についてライブラリをゼロから自前で実装する必要がある。実装すべきコードの量もそのぶん多くなるので、新しい言語のバインディングを作って貢献しようと考える人たちにとっての敷居が高くなってしまう。また、また、行指向のプロトコルはテキストデータを扱う限りは便利なのだが、スクリーンショットなど画像を扱おうとし始めると問題になる。

最初に採用した RPC の仕組みは実践的ではないということが、間もなくはっきりとしてきた。幸いにも、それ以外にもよく知られたトランスポートの仕組みがあった。幅広く採用されており、ほぼすべての言語でサポートされているという、まさに我々の望み通りのもの。それが HTTP だ。

我々は HTTP をトランスポートとして採用することに決めた。次に決める必要があったのが、エンドポイントを単一にする (SOAP 風) かあるいは複数にする (REST 風) かということだった。当初の Selenium プロトコルは単一エンドポイント方式で、コマンドと引数を符号化したクエリ文字列を使っていた。この手法はうまく機能していたが、どうも “気分的に” 間違っている感じがした。我々が思い描いていたのは、リモートの WebDriver インスタンスに

ブラウザ内から接続して、サーバーの状態を見られるようにすることだった。そして最終的に採用したのが、“REST 的な” 手法だった。複数のエンドポイント URL を用意し、HTTP メソッドを使って意味づけをする。しかし、真に RESTful なシステムに求められる制約のいくつかは満たしていない。状態を保持する場所やキャッシュなどである。その大きな理由は、アプリケーションの状態が存在する箇所はひとつだけだからである。

HTTP を採用したおかげで、符号化されたデータも (コンテンツネゴシエーションに基づく) さまざまな方法でサポートできるようになった。しかし、正式な形式をひとつ用意して、すべてのリモート WebDriver プロトコルがそれに対応できるように実装すべきだと判断した。選択肢としてすぐ候補にあがるのが、HTML や XML あるいは JSON である。XML は真っ先に候補から外した。データフォーマットとしてはよくできているしライブラリのサポートもほぼすべての言語でそろっているとはいえ、経験上オープンソースのコミュニティではあまり XML が好まれないこともわかっていた。さらに、返されるデータは共通の “形式” ではあるものの、フィールドが追加されることも十分あり得た [3]。これらの拡張機能で XML 名前空間を使ったモデリングもできたが、そんなことをしたらクライアント側のコードが無駄に複雑化してしまう。それだけは避けたかった。そこで、XML は「オプションで使うこともできる」という扱いになった。HTML はまったくもってうまい選択だと言えないだろう。我々は自前のデータフォーマットを定義しなければならない。マイクロフォーマットで無理やりフォーマットを埋め込むこともできるが、それはまるで、卵を割るときにハンマーを使うようなものだ。

最終的に残った候補が Javascript Object Notation (JSON) だった。ブラウザ側での文字列からオブジェクトへの変換は直接 `eval` を呼ぶだけでいいし、最近のブラウザなら Javascript オブジェクトと文字列の相互変換を安全に副作用なしで行うプリミティブが用意されている。現実的な観点からも、JSON はよく使われているデータフォーマットであり、ほとんどすべての言語で JSON 処理用のライブラリが用意されている。また、開発者にも人気が高い。無難な選択肢と言える。

リモート WebDriver プロトコルの第 2 イテレーションでは、HTTP をトランスポートとして採用し、UTF-8 でエンコードした JSON をデフォルトの符号化スキームとした。UTF-8 はデフォルトの符号化方式として選ばれたものであり、Unicode のサポートがあまり充実していない言語でクライアントを書くのも容易である。というのも、UTF-8 は ASCII と後方互換性があるからだ。サーバーに送信するコマンドは、URL を使ってどのコマンドが送信されたのかを判断し、コマンドへのパラメータは配列形式でエンコードする。

たとえば `WebDriver.get("http://www.example.com")` の呼び出しは、セッション ID をエンコードして最後に “/url” をつけた URL への POST リクエストにマップされる。このとき、パラメータの配列は `['http://www.example.com']` のようになる。返される結果はもう少し構造化されており、返り値やエラーコード用のプレースホルダーが用意されている。この形式は、リモートプロトコルの第 3 イテレーションまでしか続かなかった。リクエストにおけ

[3] たとえば、リモートサーバーは base64 でエンコードしたスクリーンショットに加えて発生した例外もすべて返し、デバッグの助けにしている。しかし Firefox ドライバはそれを返さない。

るパラメータの配列は、そのイテレーションで名前つきパラメータの辞書に変わった。この変更によって、デバッグ用のリクエストがとても簡単に実行できるようになった。また、クライアントがパラメータの順番を間違えてしまう可能性をなくし、システム全体としてより堅牢になった。必然的に、通常の HTTP ステータスコードを使って特定の返り値や応答を表すようになった。それが最も適切な方法だったからである。たとえば、どこにもマップされていない URL を呼ぼうとしたときや "空のレスポンス" を表したいときなどに使える。

リモート WebDriver プロトコルには二段階のエラー処理がある。無効なリクエストを扱うものと、コマンドが失敗した場合を扱うものである。無効なリクエストの例としては、サーバー上に存在しないリソースへのリクエストあるいはそのリソースが処理できないメソッドでのリクエスト (たとえば、現在のページの URL を指すリソースに対する DELETE コマンド) などがある。このような場合は、通常の HTTP 4xx レスポンスが送出される。コマンドが失敗した場合には、レスポンスのエラーコードは 500 ("Internal Server Error") となり、返すデータの中により詳しい情報を含めて何が悪かったのかをわかりやすくする。

データを含むレスポンスがサーバーから返されるときには、JSON オブジェクト形式となる。

キー	説明
sessionId	不透過なハンドル。サーバーがセッション固有のコマンドの送り先を決めるために使う。
status	コマンドの結果を表す数値のステータスコード。ゼロ以外の値は、コマンドが失敗したことを表す。
value	レスポンスの JSON データ。

レスポンスは、たとえばこのようになる。

```
{
  sessionId: 'BD204170-1A52-49C2-A6F8-872D127E7AE8',
  status: 7,
  value: 'Unable to locate element with id: foo'
}
```

見てわかるとおり、ステータスコードをレスポンス内で符号化しており、ゼロではない値が入っていることから何かがうまくいかなかったことがわかる。IE ドライバはまず最初にステータスコードを使い、プロトコル内で使う値はこの値をミラーしている。すべてのエラーコードは各ドライバで共通なので、エラー処理のコードは特定の言語で書いてすべてのドライバで共有できる。これにより、クライアント側の実装がより容易になる。

Remote WebDriver Server は単なる Java サーブレットである。これはマルチプレクサとして動作し、受け付けたコマンドを適切な WebDriver インスタンスに振り向ける。まあ大学院の二年目くらいでも書けるレベルのものだ。Firefox ドライバでもリモート WebDriver プロトコルを実装しており、そのアーキテクチャのほうがずっと興味深い。そこで、言語バインディングから受け取ったリクエストがバックエンドに到達してからユーザーに応答を返すまでの流れを追いかけてみよう。

ここでは Java を使っているものとする。また "element" が WebElement のインスタンスである。すべてはこの行からはじまる。

```
element.getAttribute("row");
```

内部的に、element は不透過な "id" を保持している。サーバーサイドではこれを使い、対話相手の要素を識別する。ここでは仮に、id の値が "some_opaque_id" であるものとして話を進める。これは Java の `Command` オブジェクトに `Map` として符号化されており、(名前付きの) パラメータ `id` に要素の ID、そして `name` に問い合わせ対象の属性の名前を保持する。

正しい URL を指すテーブル内での検索は、このようになる。

```
/session/:sessionId/element/:id/attribute/:name
```

URL のセクションの中でコロンから始まるものはすべて、変数であって後で何かの値で置き換えられるものである。パラメータ `id` と `name` は既に指定している。`sessionId` はもうひとつの不透過なハンドルで、ルーティングのために使う。これを使えば、サーバーが複数のセッションを同時に扱えるようになる (Firefox ドライバではこれができない)。この URL を展開したものは、たとえばこのようになる。

```
http://localhost:7055/hub/session/XXX/element/some_opaque_id/attribute/row
```

余談だが、WebDriver のリモートワイヤプロトコルの開発が始まったのは、URL Templates が RFC 草案として提案されたのとほぼ同時期だった。我々が考えた URL の指定方法も URL Templates も、どちらも URL 内での変数の展開 (そして派生) を許していた。残念なことに、URL Templates が提案されていることを我々が知ったのはかなり後になってからのことだった。そのため、ワイヤプロトコルの記述に URL Templates を使うことができなかった。

我々の実装するメソッドは冪等 [4] なので、ここで使うべき正しい HTTP メソッドは GET である。このあたりの処理は、HTTP を話せる Java ライブラリ (Apache HTTP Client) に委譲して、そのライブラリにサーバーを呼ばせる。

Firefox ドライバは、Firefox の拡張機能として実装されている。その基本的な設計は図 16.4 のとおりだ。多少風変わりなところがあるとすれば、それは HTTP サーバーを組み込んでいるところだ。元々は自前でこれを実装していたのだが、HTTP サーバーを XPCOM で書くというのは我々の得意分野ではない。そこで、Mozilla 自身が用意している基本的な HTTPD でそれを置き換えた。リクエストはこの HTTPD が受け取り、ほぼそのままの形で `dispatcher` オブジェクトに渡される。

ディスパッチャは、リクエストを受け取ってからサポートする既知の URL リストを順にたどり、リクエストにマッチする URL を探す。このマッチングは、クライアント側で行われた変数の置換に基づいて行われる。リクエストメソッドも含めて完全に一致するものが見つかれば、実行するコマンドを表す JSON オブジェクトを組み立てる。今回の場合は、このようなオブジェクトになる。

[4] すなわち、何度実行しても同じ値を返す。

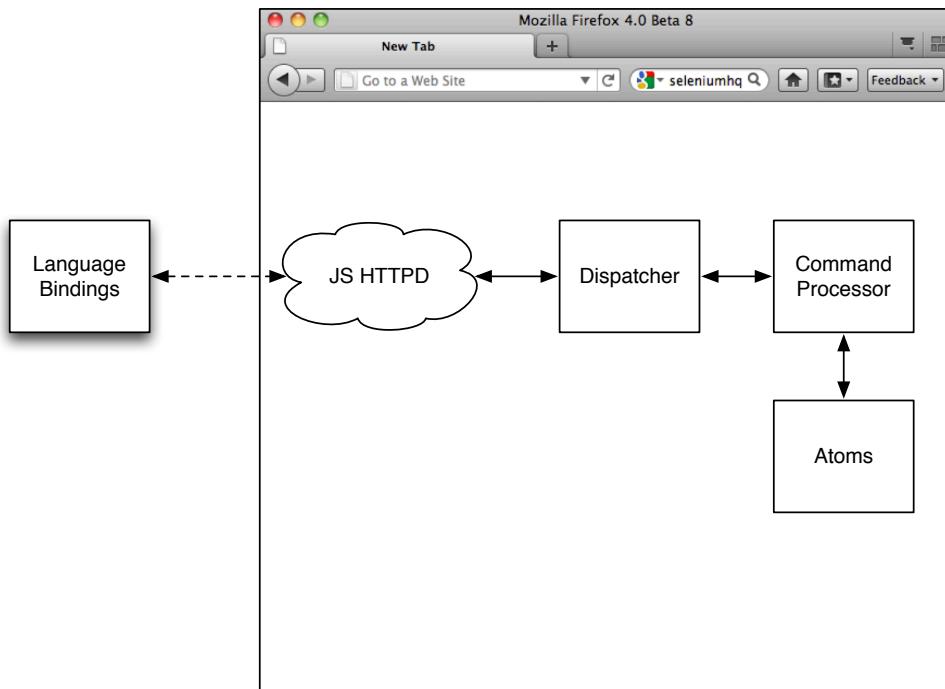



図 16.4: Firefox ドライバのアーキテクチャ概要

```
{
  'name': 'getElementAttribute',
  'sessionId': { 'value': 'XXX' },
  'parameters': {
    'id': 'some_opaque_key',
    'name': 'rows'
  }
}
```

次にこれが、JSON 文字列として我々の書いた XPCOM コンポーネントに渡される。このコンポーネントは CommandProcessor と呼ばれている。このようなコードだ。

```
var jsonResponseString = JSON.stringify(json);
var callback = function(jsonResponseString) {
  var jsonResponse = JSON.parse(jsonResponseString);

  if (jsonResponse.status != ErrorCode.SUCCESS) {
    response.setStatus(Response.INTERNAL_ERROR);
  }

  response.setContentType('application/json');
  response.setBody(jsonResponseString);
```

```
  response.commit();
};

// コマンドをディスパッチする
Components.classes['@googlecode.com/webdriver/command-processor;1'].
    getService(Components.interfaces.nsICommandProcessor).
    execute(jsonString, callback);
```

やたら大量のコードがあるが、ポイントとなるのは次のふたつだ。まず、上のオブジェクトを JSON 文字列に変換する。次に、コールバックを渡してメソッドを実行させる。これが、送出する HTTP レスポンスを作る。

コマンドプロセッサで実行するメソッドは、"name" を見てどの関数を呼ぶかを判断し、そしてそれを実行する。この実装関数に渡す最初のパラメータは `respond` オブジェクトで (このように呼ばれているのは、もともとこの関数は単にユーザーにレスポンスを返すためだけのものだったからである)、これは返す値をカプセル化するだけではなく、レスポンスをユーザーに送り返せるようにするメソッドや DOM の情報を見つけるための仕組みも用意されている。二番目のパラメータは、先述の `parameters` オブジェクトの値 (この場合は `id` と `name`) である。この方式のメリットは、各関数が統一インターフェイスを持っていて、それがクライアント側で使うデータ構造を反映しているということだ。つまり、コードを書くときに頭の中で考えるモデルがクライアント側でもサーバー側でも同じようになるのだ。ここに `getAttribute` の実装を示す。これは先ほど 16.5 節で見たものである。

```
FirefoxDriver.prototype.getElementAttribute = function(respond, parameters) {
  var element = Utils.getElementAt(parameters.id,
                                   respond.session.getDocument());
  var attributeName = parameters.name;

  respond.value = webdriver.element.getAttribute(element, attributeName);
  respond.send();
};
```

要素の参照に矛盾を生じさせないようにするために、最初の行は単にキャッシュ内の不透過な ID が参照する要素を探すだけになっている。Firefox ドライバの場合、この不透過な ID は UUID であり、"キャッシュ" は単なるマップである。`getElementAt` メソッドは、要素への参照が既知のものであるかどうかとそれが DOM にアタッチされているかどうかもチェックする。どちらかのチェックに失敗すると、その ID は (必要に応じて) キャッシュから削除され、例外を投げてそれをユーザーに返す。

最後から二行目では、先述のブラウザ自動化用 atom を使っている。ここでは一つのスクリプトとしてコンパイルされ、拡張機能の一部として読み込まれている。

最後の行で呼ばれているのが `send` メソッドだ。このメソッドはシンプルなチェックを行い、execute メソッドで指定したコールバックを呼んでからレスポンスを送出する。レスポンスは JSON 文字列形式でユーザーに戻され、それがこのような形式のオブジェクトに移される (`getAttribute` の返り値が "7"、つまりその要素が見つからなかったものと仮定する)。

```
{
  'value': '7',
  'status': 0,
  'sessionId': 'XXX'
}
```

その後、Java クライアントが status フィールドの値をチェックする。もし値がゼロでなければ、数値のステータスコードを適切な型の例外オブジェクトに変換して投げる。その際に、"value" フィールドの値を使ってユーザー向けのメッセージを設定する。status がゼロの場合は、"value" フィールドの値をユーザーに返す。

ほとんどは、ごく自然な処理だろう。しかし一カ所だけ、賢明な読者なら疑問に思うところがあるはずだ。なぜディスパッチャは、`execute` メソッドを呼ぶ前にわざわざオブジェクトを文字列に変換するのだろう?

なぜそうしているかというと、Firefox ドライバは Javascript だけで書かれたテストの実行もサポートしているからである。普通は、これをサポートするのはかなり難しい。テストが実行されるのはブラウザの Javascript セキュリティサンドボックスの中であり、テストで有用な多くの作業 (別ドメインへの移動やファイルのアップロードなど) に制限が出てしまうからである。しかし Firefox の WebDriver 拡張機能では、サンドボックスから脱出するためのハッチを提供している。document 要素に `webdriver` というプロパティを追加しているのだ。WebDriver の Javascript API はこれを見て、次のような操作ができると判断する。JSON でシリアライズしたコマンドオブジェクトを document 要素の `command` プロパティに追加して `webdriverCommand` イベントを発火させ、何らかの要素での `webdriverResponse` イベントの発生を監視する。このイベントが、`response` プロパティの値が設定されたことを示す。

これはつまり、WebDriver 拡張機能をインストールした Firefox でウェブをブラウズするのは非常に危険だということだ。悪意のある人なら、リモートから容易にブラウザを乗っ取れてしまうからである。

その裏側では DOM メッセンジャーが動いていて `webdriverCommand` を待機している。シリアライズした JSON オブジェクトをこれが読み込み、コマンドプロセッサの `execute` メソッドを呼び出す。このとき、コールバックは単に document 要素の `response` 属性に設定されるだけである。それが、期待する `webdriverResponse` イベントを発火させる。

16.7 IE ドライバ

Internet Explorer は興味深いブラウザである。さまざまな COM インターフェイスが一斉に動く構造になっている。Javascript エンジンもまったく同じで、なじみのある Javascript の変数も実際はその背後にある COM インターフェイスへの参照となっている。たとえば Javascript の `window` の実体は `IHTMLWindow` である。つまり、`document` は COM インターフェイス `IHTMLDocument` のインスタンスとなる。Microsoft は、既存のふるまいを維持しつつブラ

ウザに機能を追加するというすばらしい仕事をやってのけた。つまり、IE6 が公開する COM クラスを使ったアプリケーションは、そのまま IE9 でも動くということである。

Internet Explorer ドライバのアーキテクチャは、時を経て成長してきた。その設計の根底にある大きな目標は、インストーラーを使わないということだ。あまり聞き慣れない要件だと思うので、ここで少し捕足しておこう。インストーラーを不要にしたい第一の理由は、WebDriver が “5 分でテストできる” という目標を満たせなくなるからだ。開発者がパッケージをダウンロードしてインストーラーを実行するまでに、ある程度の時間を要してしまう。さらに重要なのは、WebDriver のユーザーの中にはは自分のマシンにソフトウェアをインストールする権限を持っていない人も比較的多いということである。インストーラーをなくせば、あるプロジェクトで IE を使ったテストを始めるときに、継続的インテグレーションサーバーにログオンしてインストーラーを実行するという必要もなくなる。最後に、インストーラーを実行するという習慣を持たない言語もあるということだ。たとえば Java なら、通常は CLASSPATH の通った場所に JAR を置くだけけだ。経験上、インストーラーがついているライブラリはあまり好まれず、使われることもない。

だから、インストーラーはやめた。その結果、次のようになった。

Windows 上でのプログラミングに使う言語として自然な選択は、.Net 上で動くもの (おそらく C#) になるだろう。IE ドライバは IE の COM オートメーションインターフェイスを使っており、IE 自体と密に結合している。IE の COM オートメーションインターフェイスは、すべてのバージョンの Windows に組み込まれている。特に使っているのはネイティブの MSHTML および ShDocVw DLL であり、これらは IE の一部である。C# 4 より前のバージョンでは、CLR/COM の相互運用性は Primary Interop Assembly (PIA) を分離することで実現していた。PIA は本質的に、CLR が管理する世界と COM が管理する世界の橋渡しをするために作られたものである。

残念なことに、C# 4 を使おうとすると.Net ランタイムの新しいバージョンを使うことになってしまう。多くの企業では最先端を使うのを避け、問題は残っているが安定している旧リリースを使うことを好む。C# 4 を使ってしまうと、無視できない割合のユーザー層を排除してしまうことになる。PIA を使うことにはそれ以外のデメリットもある。ライセンスの制約を考えてみよう。Microsoft に問い合わせた結果明らかになったことは、Selenium プロジェクトには MSHTML や ShDocVw の PIA を配布する権限がないということだった。仮に配布する権利が認められたとしても、世間にインストールされている Windows と IE のライブラリには無数の組み合わせがある。それらすべてに対応する PIA を配布しなければならないということだ。クライアントマシン上で、その場で PIA を作るという方法もあるが、それは使えない。開発者向けのツールが必要になるし、通常のユーザーのマシン上にはそんなものはインストールされていないからだ。

したがって、大規模なコードを書くには C#はとても魅力的な言語だが、今回の選択肢からは消えた。我々が必要としたのは何かネイティブな言語で、少なくとも IE との通信ができるものだった。この目的にかなう次の選択肢は C++であり、最終的に我々が選んだのもこの言語だった。C++を使えば PIA が不要になるというメリットがあるが、それと同時に、Visual

Studio C++ランタイム DLL を再配布するかあるいは静的にリンクするかの選択を迫られることになってしまう。DLL を配布するにはインストーラーが必要になってしまうので、IE と通信するライブラリを静的にリンクすることにした。

インストーラーを使わないという要件を満たすために、かなりのコストがかかっている。しかし、当初のテーマであった「複雑性をどこに押し込めるか」を考えると、ユーザーにとって使いやすくするための投資としては十分に意味のあるものだ。C++を採用するという決断については、現時点で改めて評価しなおしているところである。というのも、ユーザーにとっての使いやすさと引き替えに、協力してくれる開発者の母数が減ってしまうという事実があるからだ。高度な C++で書かれたオープンソースプロジェクトに貢献してくれる可能性のある開発者の数は、C#のプロジェクトに比べて大幅に少ないように感じる。

IE ドライバの当初の設計を図 16.5 に示す。

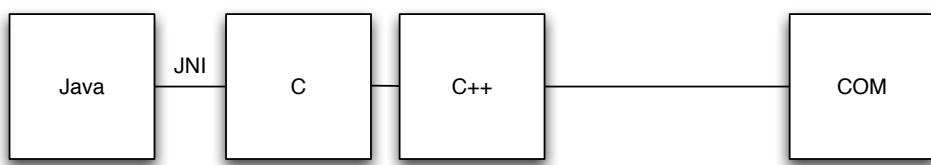



図 16.5: 当初の IE ドライバ

スタックの最下層から話を始めよう。ここでは、IE の COM オートメーションインターフェイスを使っていることがわかる。概念的な意味での使いやすさを向上させるために、この生のインターフェイスを C++のクラス群でラップした。このラッパーは、WebDriver API の構造に似せてある。Java のクラスに C++との通信をさせるために JNI を使い、JNI のメソッドで COM インターフェイスの C++ラッパーと通信している。

このアプローチは、クライアント側の言語が Java だけに限られる場合はうまくいく。しかし、サポートすることになったすべての言語に対してバックにあるライブラリを変更するのは大変だし、複雑になってしまう。したがって、JNI だけでもうまくいくとは言え、これだけではまだ適切な抽象化が実現できていない。

適切な抽象化とは何だったのか? 我々がサポートしようとしていたすべての言語には、C のコードを直接呼び出す仕組みがあった。C#の場合は PInvoke 形式がそれにあたる。Ruby には FFI があるし、Python には ctypes がある。Java の世界には、すばらしいライブラリである JNA (Java Native Architecture) が存在する。我々が必要としていたのは、これらの共通項を使う API を公開することだった。これを実現するためにオブジェクトモデルを平坦化し、シンプルな 2 文字か 3 文字のプレフィックスを使って各メソッドの “ホームインターフェイス” を示した。たとえば “wd” は “WebDriver” を表し、“wde” は WebDriver Element を表す。つまり `WebDriver.get` は `wdGet` となり、`WebElement.getText` は `wdeGetText` となった。各メソッドはステータスコードを表す整数値を返し、同時に “out” パラメータを使ってそれ以外のデー

タも返せるようにした。最終的に、メソッドのシグネチャはこのようになる。

```
int wdeGetAttribute(WebDriver*, WebElement*, const wchar_t*, StringWrapper**)
```

呼び出す側のコードでは、WebDriver や WebElement そして StringWrapper は不透過型となる。API ではこれらの違いを表し、どの値をどのパラメータで使うのかを明確にしている。しかし、単に “void *” とすることもできる。また、テキストにはワイド文字を使っていることもわかるだろう。これは、国際化したテキストを適切に扱えるようにするためである。

Java 側では、この関数のライブラリをインターフェイス経由で公開している。それを使って WebDriver が用意する通常のオブジェクト指向のインターフェイスと同じようにすることができる。たとえば、Java での getAttribute メソッドの定義はこのようになる。

```
public String getAttribute(String name) {
  PointerByReference wrapper = new PointerByReference();
  int result = lib.wdeGetAttribute(
      parent.getDriverPointer(), element, new WString(name), wrapper);

  errors.verifyErrorCode(result, "get attribute of");

  return wrapper.getValue() == null ? null : new StringWrapper(lib, wrapper).toString();
}
```

これが、図 16.6 で示す設計につながる。

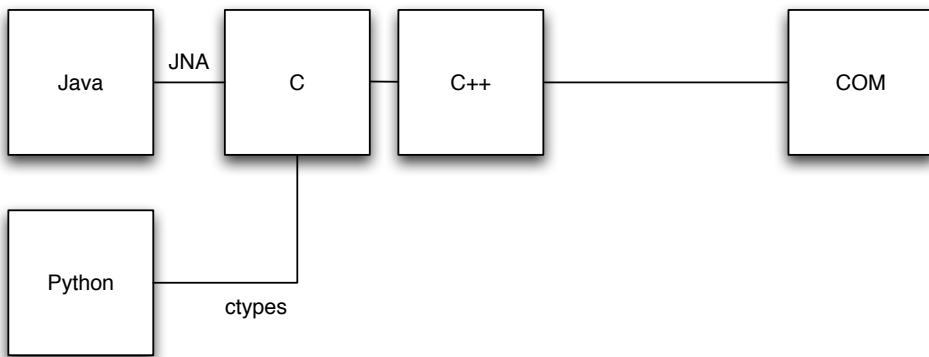



図 16.6: 手を加えられた IE ドライバ

すべてのテストをローカルマシンで動かしている間はこれでうまくいく。しかし、IE ドライバをリモートの WebDriver で使い始めると、ランダムなロックが発生してしまうようになった。原因を追跡していくと、最後は IE の COM オートメーションインターフェイスの制約に行き着いた。このインターフェイスは、“Single Thread Apartment” モデルで使うように作られていたのだ。これは本質的に、インターフェイスを毎回同じスレッドから呼び出す必

要があるということである。ローカルで動かしている場合は、デフォルトでこの挙動となる。しかし Java アプリケーションサーバーは、負荷に対応するために複数スレッドをまとめて使う。で、どうなったって? どんな場合にも同じスレッドを使って IE ドライバにアクセスさせる方法なんか見つからなかった。

この問題に対するひとつの解決策が、IE ドライバをシングルスレッドのエグゼキュータ内で実行してアプリケーションサーバー内の Future 経由のアクセスをすべてシリアライズすることで、しばらくの間は我々もその方法をとっていた。しかしこれは、複雑性を呼び出し側のコードに残してしまうという点でアンフェアであり、不注意で IE ドライバを複数のスレッドから呼んでしまうという自体が容易に想像できた。そこで、この複雑性はドライバ自身に押し込めることに決めた。IE のインスタンスを個別のスレッドに持たせ、Win32 API の `PostThreadMessage` を使ってスレッド越しの通信を行った。というわけで、執筆時点での IE ドライバの設計は図 16.7 のようになっている。

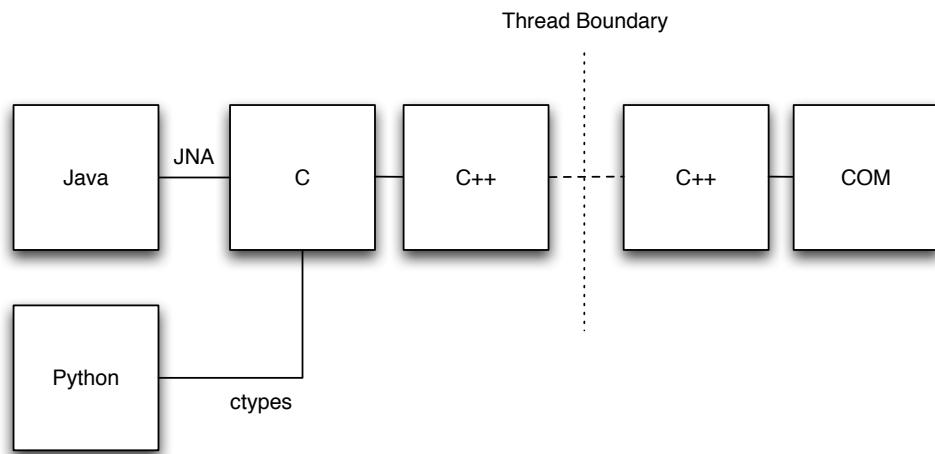



図 16.7: Selenium 2.0 alpha 7 の時点の IE ドライバ

これは自ら進んで選んだ設計ではない。しかしうまく動いており、ユーザーに不便な思いをさせずに済んでいる。

この方式の問題のひとつが、IE のインスタンスがロックされているかどうかを自分では判断しづらいことだ。これが問題となるのは、DOM を操作している際にモーダルダイアログが開いたりスレッド境界のはるか向こう側で致命的な障害が発生したりした場合である。その対策として、送信するすべてのスレッドメッセージにはタイムアウトを設定しており、その値は太っ腹にも 2 分となっている。メーリングリストでのユーザーの声を見る限り、この値はほぼ適切なようだ。しかし、常にこれが正解というわけではないので、IE ドライバの今後のバージョンではタイムアウトの値を変更可能にする予定だ。

もうひとつの問題は、内部的なデバッグが非常に難しくなるということだ。まずスピード

を問われる (要するに、2 分以内にコードを追いかけきらないといけない) し、適切なブレークポイントを設定して、スレッドをまたがるコードの流れをきちんと追いかけなければならない。言うまでもないが、オープンソースの世界には興味をそそられる課題が満載だ。こんな汚れ仕事を進んでやろうとする人はまずいないだろう。そのおかげでシステムのバス係数が劇的に低下してしまう。プロジェクトのメンテナとして、これは気がかりなことだ。

その対策として、IE ドライバをどんどん Automation Atoms 化している。Firefox ドライバや Selenium Core と同様に、だ。使おうとしている atom をコンパイルして C++のヘッダファイルを作り、個々の関数を定数として公開する。実行時には、Javascript を使ってこれらの定数を実行する。このようにすると、IE ドライバのコードの大部分は C コンパイラがなくても開発・テストできるようになる。そのぶん、バグを見つけたり修正してくれたりする人たちにとっても敷居が低くなるだろう。最終的な目標は、インタラクション API だけをネイティブコードのまま残し、それ以外はできる限り atom に任せるようにすることだ。

もうひとつのアプローチとして検討中なのが、IE ドライバを書き直して軽量 HTTP サーバーを使うようにするということだ。そうすれば、IE ドライバをリモート WebDriver として扱えるようになる。もしこれが実現すれば、スレッドの境界にまつわる複雑性の多くを取り除け、必要なコードの量も減らせるうえに、処理の流れも非常に追いやすくなる。

16.8　Selenium RC

特定のブラウザとの密な結合が常に可能だとは限らない。そんな場合、WebDriver は次善の策としてもともと Selenium が使っていた仕組みを利用する。つまり Selenium Core を使うということだ。これはピュア Javascript フレームワークであり、Javascript サンドボックスのコンテキストで動くこともあっていろいろ制約も多い。WebDriver の API を使う側から見ると、サポート対象とされているブラウザの中にもいくつかのレベルがあるということだ。いくつかのブラウザは密に統合されているので期待通りの制御ができるが、そうでないブラウザは Javascript によるサポートしかなく、もともとの Selenium RC が持っていた機能と同程度の制御しかできないということになる。

概念的には、ここで採用する設計は図 16.8 のように極めてシンプルなものである。

ご覧の通り、全体が大きく三つに分かれている。クライアントのコード、中間サーバー、そしてブラウザ内で動作する Selenium Core の Javascript コードだ。クライアント側は単なる HTTP クライアントであり、シリアライズしたコマンドをサーバー側に送る。リモート WebDriver とは違って単にエンドポイントがひとつあるだけであり、どの HTTP メソッドを使うかはあまり関係ない。その理由のひとつは、Selenium RC のプロトコルが Selenium Core のテーブルベースな API を元にしていることだ。そのため、URL のクエリパラメータを三つ使うだけで API 全体をカバーできてしまう。

クライアントが新しいセッションを開始すると、Selenium Server はリクエストされた “ブラウザ文字列” に対応するブラウザランチャーを探す。そしてランチャーが、指定されたブ

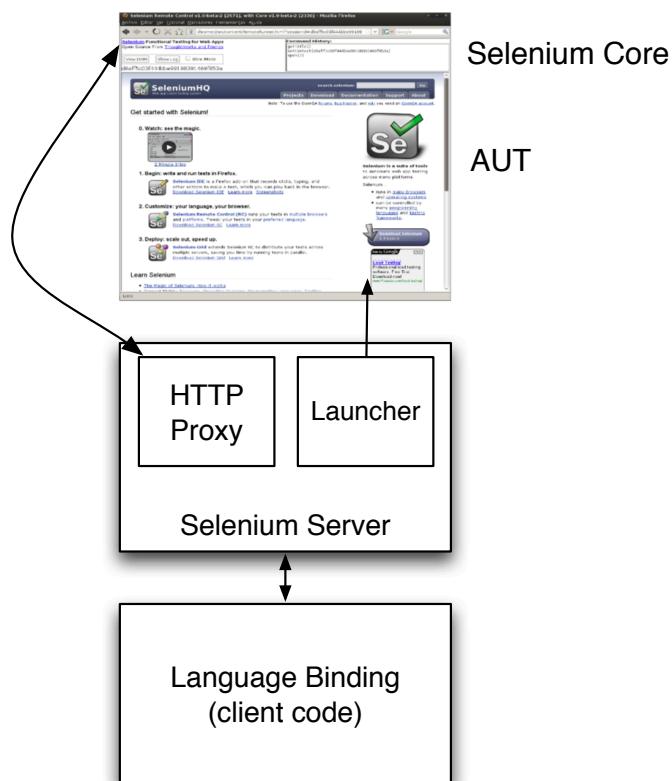



図 16.8: Selenium RC のアーキテクチャの概要

ラウザの構成を設定してブラウザのインスタンスを立ち上げる。Firefox の場合は、事前に用意したプロファイルやプレインストールの拡張機能 (“quit” コマンドを処理するための拡張や、“document.readyState” に対応する拡張など。“document.readyState” は、Selenium がサポートする Firefox のバージョンの中でも古いものには存在しない) を展開するだけのことだ。ここで行われる設定の中でもポイントとなるのが、Selenium Server が自分自身をブラウザのプロキシに設定するということだ。これにより、少なくとも一部のリクエスト (“/selenium-server” に向けたもの) を Selenium Server でルーティングできるようになる。Selenium RC は、次の三つのモードのうちのいずれかで動作する。一つのウィンドウ内のフレームを制御するモード (“singlewindow” モード)、別のウィンドウを開いて AUT をそのウィンドウで制御するモード (“multiwindow” モード)、そして自分自身をプロキシ経由でページに差し込むモード (“proxyinjection” モード) だ。処理モードによっては、すべてのリクエストがプロキシ経由となることがある。

ブラウザの設定を終えたらブラウザを立ち上げる。そのときの初期 URL は、Selenium Server が稼働するページ—`RemoteRunner.html` である。このページがすべての初期化処理を受け持

ち、Selenium Core に必要な Javascript ファイル群をすべてここで読み込む。読み込みが完了すると、“runSeleniumTest” 関数が呼ばれる。この関数は、Selenium オブジェクトのリフレクションを使って利用可能なコマンド群を初期化をしてから、メインのコマンド処理ループを起動する。

ブラウザ内で動く Javascript が XMLHttpRequest を待ち受けサーバーの URL(/selenium-server/driver) に送る。このサーバーがすべてのリクエストのプロキシとなり、リクエストが正しい場所に届くことを保証する。リクエストを送るだけではなく、その前に前回実行したコマンドのレスポンスを送信したり、ブラウザが立ち上がったときに “OK” を送ったりもする。サーバーはその後リクエストをオープンし続け、クライアントからユーザーのテストのコマンドを受け取ったら、オープンしていたリクエストのレスポンスを Javascript に返す。この方式は俗に “Response/Request” と呼ばれていたが、最近では “Comet with AJAX long polling” と呼ばれることが多い。

なぜ RC の挙動がそうなっているのかって? Server をプロキシとして設定しなければならない理由は、Javascript の “同一生成元” ポリシーに違反せずにすべてのリクエストを横取りするためだ。“同一生成元” ポリシーとは、Javascript ではスクリプトがある場所と同じサーバー上のリソースしかリクエストできないという制限のことである。そもそもこれはセキュリティを考慮して用意されたポリシーだが、ブラウザ自動化フレームワークの開発者から見ると極めて邪魔なものであり、こうでもしないとどうにもならないのだ。

XmlHttpRequest コールをサーバーに向けて行う理由は次のふたつだ。まずは最も重要な理由から。HTML5 の一部である WebSockets が大半のブラウザに実装されるようになるまでは、サーバープロセスをブラウザ側から立ち上げるための信頼できる方法が存在しないということである。つまり、サーバーはどこか別の場所で立ち上げないといけないということだ。もうひとつの理由は、XMLHttpRequest がレスポンスコールバックを非同期で呼び出すことだ。これはつまり、次のコマンドを待ち受けている間にも、操作中のブラウザの動きには何の影響も及ぼさないと言うことである。次のコマンドを待ち受ける方法は、それ以外にも二通り存在する。ひとつは、定期的にサーバーをポーリングして、何か別のコマンドが実行されていないかを調べる方法だ。しかしこの方法だと、ユーザーのテストにある程度の遅延が生じてしまう。もうひとつの方法は、Javascript をビジーループの中に置いてしまうことだ。しかしこの方法は CPU を食いつぶしてしまい、他の Javascript をブラウザ内から実行できなくなってしまう (ひとつのウィンドウ内のコンテキストには、Javascript のスレッドをひとつしか実行できないからである)。

Selenium Core の中は、大きく二つの部分に分かれている。本体の selenium オブジェクトは利用可能なすべてのコマンドのホストとして機能し、その API をユーザー向けに提供する。もうひとつの部分が browserbot だ。これは、Selenium オブジェクトが各ブラウザの差異を取り払うための抽象化として使うもので、一般的なブラウザの機能を理想的な状態で表している。これによって selenium の関数はよりきれいで保守しやすくなり、一方で browserbot にすべてを集中させることになる。

Selenium Core も、徐々に Automation Atoms を使うように書き換えているところだ。selenium

と browserbot はおそらく残さざるを得ないだろう。大量のコードがこれらの公開する API に依存しているからである。しかし、最終的にはこれらも単なるガワだけのクラスにしてしまって、内部の実装はすぐにでも atoms に委譲させてしまいたいところだ。

16.9 過去を振り返って

ブラウザの自動化フレームワークを作るという作業は、部屋の塗装に似ている。始める前にはとても簡単な作業に思えるのだ。「要するに、ざっと塗ってしまえばそれでおしまいでしょ?」実際はそうではなく、作業をすればするほどやるべきことが増えてきて、ひとつひとつのタスクが面倒なものになっていく。部屋の塗装の例だと、照明周りや幅木などを手掛け始めるととたんに時間がかかるようになる。ブラウザの自動化フレームワークでこれにあたるのが、ブラウザごとの機能の違いや微妙な動きだ。そのおかげで作業はさらに複雑になる。極めつけが、私の隣の席で Chrome ドライバを開発するダニエル・ワグナー=ホールだ。あまりにもイライラした彼は、デスクに両手をたたきつけてこう叫んだ。「なんでこんなあり得ないことばかり起こるんだよ!」できることなら過去にさかのぼって当時の自分にこう伝えてやりたい。「そのプロジェクト、たぶん思ってるよりもずっと時間がかかるよ」と。

今さらどうにもならないことではあるが、もし Automation Atom のようなレイヤーの必要性をもっと早くに認識して対応していたらどうなっただろうかと考えることもある。きっと、これまでに我々が直面してきた問題 (内部的な問題も外部的な問題も、そして技術的な問題も社会的な問題も) のいくつかは、もっと容易に対処できたことだろう。Core や RC は、特定の言語—基本的に Javascript と Java—に注力しすぎた実装になっている。ジェイソン・ハギンズはかつて、これを指摘して Selenium の “ハッカビリティ” を改善し、そのおかげでプロジェクトに参加するための敷居が下がった。atom があったからこそ、WebDriver が幅広いハッカビリティを確保できたのだ。我々が atom を幅広く適用できたのは、Closure コンパイラのおかげである。Closure コンパイラがオープンソースでリリースされるとすぐに、それを採用した。

「こうすればよかった」だけではなく、「これはうまくできた」ということを振り返るのもいいだろう。フレームワークを書くときにユーザーの視点を重視したという判断は、今でも間違いではなかったと思っている。初期の段階では、改善すべき点をアーリーアダプターが指摘してくれたおかげで、ツールの使いやすさを急速に改善することができた。後に、WebDriver がより高度な作業を行うようになるにつれて、利用する開発者の数も増えてきた。新たな API を追加するときには今まで以上に注意を払うようになり、それがプロジェクトを引き締めることにもつながった。我々がやろうとしていることを考えると、これは非常に大切なことだ。

ブラウザと密に結合させたことには、功罪の両面がある。よい面は、忠実にユーザーをエミュレートしてブラウザを完璧に制御できるようになったということだ。逆に悪い面は、この手法をとると技術的な要求が厳しくなるということだ。特に、ブラウザのフックポイントを見つけ出すのが大変になる。その大変さは、IE ドライバの開発の進み具合を見ればよくわ

かる。ここでは紹介できなかったが Chrome ドライバも同様で、これもまた話せば長くなる経緯がある。いつの日か、この複雑性とうまく向き合う方法を見つけ出したいものだ。

16.10　今後に向けて

WebDriver との密な結合ができないブラウザってのが、常に存在する。なので、Selenium Core は今後もいつだって必要になるだろう。当初から続く現在の設計を変更して、atoms も使っている Closure Library と同じものを使うように変更しようという動きが進行中である。また、既存の WebDriver の実装にも、atoms をより深く埋め込んでいこうという動きもある。

WebDriver の当初の目標のひとつが、他の API やツールを組み立てるためのブロックとして使えるようにすることだった。もちろん、Selenium が唯一のブラウザ自動化ツールというわけではない。それ以外にも、オープンソースのブラウザ自動化ツールはいくらでも存在する。そのひとつが Watir (Web Application Testing In Ruby) であり、Selenium と Watir の開発者が協力して、Watir API を WebDriver のコアにのせようという動きも始まっている。我々は、他のプロジェクトとの協力を望んでいる。すべてのブラウザに対応し続けるという作業はたいへんなものだからである。しっかりとした中核があって、他の開発者たちがその上に何かを構築しやすいようにできれば、すばらしいことだろう。我々としては、その中核が WebDriver となって欲しい。

そんな将来をうかがわせるような申し出が Opera Software からあった。彼らは独自に WebDriver API を実装し、WebDriver のテストスイートを使ってそのコードのふるまいを検証し、そして OperaDriver としてリリースする。Selenium チームのメンバーは Chromium チームとも共同作業をしており、WebDriver のサポートやよりよいフックを Chromium に追加しようとしている。Chrome に対しても同様だ。Mozilla とも良好な関係を築いている。彼らは FirefoxDriver のコードに貢献してくれたし、あの有名な Java ブラウザエミュレータである HtmlUnit の開発者も提供してくれた。

ひとつの見方として、この傾向は今後も続くだろう。さまざまなブラウザで、統一された方法で自動化のフックができるようになるという流れだ。ウェブアプリケーションのテストを書く開発者にとってのメリットは明白だし、ブラウザを作る側にとってもその利点は明らかだ。たとえば、手動テストのコストと比較して、多くの大規模プロジェクトでは自動テストにより依存している。もし特定のブラウザでのテストが不可能 (あるいはばかみたいに高いコストがかかる) なら、そのブラウザに対するテストは行われないだろう。アプリケーションが複雑になればなるほど、テストしていないブラウザでそれがうまく動くかどうかは怪しくなる。最終的に統一された自動化フックが WebDriver ベースのものになるかどうかはわからない。でも、そうあって欲しいものだね!

今後数年の動きが楽しみだ。我々はオープンソースプロジェクトなので、いつでもみなさんの参加を歓迎する。さあ、一緒に `http://selenium.googlecode.com/` への旅に出ないかい?

第 17 章

Sendmail

Eric Allman

電子メールのプログラムと聞いて、たいていの人が思い浮かべるのはメールクライアントだろう。厳密には、Mail User Agent (MUA) と呼ばれているものだ。しかし、電子メールを扱うソフトウェアには、もうひとつ重要なものがある。それが、実際に送信者から受信者にメールを配送するソフトウェア—Mail Transfer Agent (MTA) である。インターネット上で最初に登場した MTA であり、現在でも幅広く使われているものが sendmail だ。

Sendmail が最初に作られたのは、まだインターネットが公式には存在しなかったころである。そして、それから大成功を収めた。1981 年に最初に作られたころ、インターネットはまだ学術的な実験段階で、接続されているホストはたかだか数百台に過ぎなかった。それが今や、2011 年 1 月の時点でインターネットに接続されているホストは 8 億を超える [1]。当時、今のこの状況を想像できた人はあまりいなかっただろう。Sendmail は、インターネット上の SMTP の実装として、今でも最も使われているものである。

17.1　むかしむかし…

後に sendmail として知られることになるプログラムの最初のバージョンが書かれたのは 1980 年のことだった。もともとは、メッセージを別のネットワークに転送するためのちょっとしたハックだった。インターネットは構築されていたが、当時はまだそれほど機能的ではなかった。実際のところ、当時はさまざまなネットワークがコンセンサスを得ないままに提案されていた。アメリカでは Arpanet が使われており、インターネットはその上位版として設計された。しかしヨーロッパでは OSI (Open Systems Interconnect) の取り組みを支持しており、おそらく OSI のほうが勝ちを収めるだろうとみられていたこともあった。どちらも電話会社から借りた専用線を使っており、米国での速度は 56 Kbps だった。

おそらく、(接続するマシンやユーザーの数で考えると) 当時最も成功していたネットワークは UUCP ネットワークだろう。このネットワークの特徴は、中央管理型の権限がどこにも

[1] http://ftp.isc.org/www/survey/reports/2011/01/

なかったことだ。ある意味では、ピアツーピアのネットワークの元祖と言えるだろう。このネットワークは電話回線でのダイアルアップで動いており、最大で 9600 bps の速度しか出なかった。その当時の最速のネットワーク (3 Mbps) は Xerox の Ethernet をベースとしたもので、これは XNS (Xerox Network Systems) というプロトコルで動いていた—しかし、ローカルにインストールした環境以外では動作しなかった。

当時の環境は、今とは異なっていた。コンピュータはそれぞれ異質なものであり、そもそも 1 バイトを 8 ビットにするかどうかさえ完全には合意されていなかった。たとえば PDP-10(1 ワードが 36 ビット、1 バイトが 9 ビット)、PDP-11(1 ワードが 16 ビット、1 バイトが 8 ビット)、CDC 6000 シリーズ (1 ワードが 60 ビット、1 文字が 6 ビット)、IBM 360(1 ワードが 32 ビット、1 バイトが 8 ビット)、そして XDS 940 や ICL 470、Sigma 7 などがあった。そのころ赤丸急上昇中だったプラットフォームのひとつが Unix で、これはベル研究所が発表したものだった。Unix ベースのマシンの大半は 16 ビットアドレス空間を保持していた。当時の Unix マシンの主流は PDP-11 で、Data General 8/32 や VAX-11/780 が登場してきたころだった。スレッドはまだ存在しなかった—実際のところ、ダイナミックプロセスという概念自体がまだまだ出たてのものだったのだ (Unix にはスレッドがあったが、IBM の OS/360 みたいな "本物の" システムにはまだ実装されていなかった)。ファイルのロックはまだ Unix カーネルではサポートされていなかった (ファイルシステムのリンクを使うという裏技はあった)。

とりあえずあるにはあったものの、ネットワークは一般的に低速だった (その多くは 9600 ボーの TTY 回線を使っていた。お金持ちの中には Ethernet を使っているところもあったが、それでもローカルネットワークだけの話だった)。あの由緒あるソケットインターフェイスが発明されるのは、それから何年か後のことである。公開鍵による暗号化の仕組みもまだ発明されていなかったので、ネットワークセキュリティに関して我々の知る仕組みのほとんどは実現不可能だった。

ネットワーク上を流れる電子メールは Unix で既に実現されていたが、そのためにはちょっとしたハックを要した。当時のユーザーエージェントとして一番よく使われていたのが `/bin/mail` コマンド (現在では `binmail` や `v7mail` と呼ばれることもある) だが、それ以外のユーザーエージェントを使っているところもあった。たとえばバークレーの `Mail` は、メッセージを個別のアイテムとして扱う方法を知っていた。単に `cat` プログラムに任せるだけではなかった。どのユーザーエージェントも、`/usr/spool/mail` ディレクトリを直接読んだり (そして直接書き込んだり!) していた。実際に格納されるメッセージを抽象化することなどなかったのだ。

メッセージをネットワークに送るかローカルの電子メールに送るかの振り分けロジックは、単にアドレスを見て中に感嘆符 (UUTP の場合) あるいはコロン (BerkNET の場合) が含まれているかどうかを見るだけのことでしかなかった。Arpanet にアクセスする人たちは、完全に個別のメールプログラムを使う必要があった。これは他のネットワークとの相互運用ができず、ローカルメールも別の場所に異なる書式で保存していた。

さらにおもしろいのが、この時点ではまだ、メッセージ自体の書式についても事実上標準化されていなかったということだ。大まかに決まっていたのは、メッセージの先頭にヘッダフィールドを置き、各ヘッダフィールドの後には改行を置いて、フィールド名とその値はコ

ロンで区切るということくらいだった。それ以外のこと、たとえばどんなヘッダフィールド名を使うかや個々のフィールドの構文などについてはほとんど標準化されていなかった。たとえば`Subject:`の代わりに`Subj:`を使うシステムもあったし`Date:`フィールドの構文もそれぞればらばらだったし、システムによっては`From:`フィールドのフルネームを解釈できないものもあった。さらに、ドキュメントもあいまいなものばかりだったし、ドキュメントが実際の内容と食い違っていることもあった。特に RFC 733(Arpanet メッセージについての説明)は実際に使われている内容と微妙に異なっていた (しかし重要な違いであることもあった)。そして、実際のメッセージ送信の仕組みは実際のところ公式には文書化されていなかった (その仕組みを扱う RFC はいくつかあったが、きちんとした定義はどこにもなかった)。メッセージングシステムまわりはこんなひどい状況だった。

1979 年、INGRES Relational Database Management Project(私の昼間の業務だった) は DARPA の資金援助を受け、9600bps の Arpanet 接続を我々の PDP-11 につなげることになった。その当時、コンピュータサービス部門で Arpanet に接続しているマシンはそれだけだったので、誰もが我々のマシンを使って Arpanet を使いたがった。しかし、そのマシンはすでに最大限に使い切っており、使えるログインポートはあとふたつしか残っていなかった。そこで、そのふたつを部門内で共有した。そのため、頻繁に争いが発生した。しかし、私は気づいた。我々の大半はリモートログインやファイル転送を求めているのではなく、単に電子メールが使いたいだけだったのだ。

そんなところに、sendmail(当初は delivermail という名前だった) が登場した。このカオスを何とか統一しようという試みだ。すべての MUA(Mail User Agent、あるいはメールクライアント) は、単に delivermail を呼び出すだけでメールを配送することができた。いちいち調査をしてアドホックな (そして、たいてい互換性のない) 対応をする必要もない。Delivermail/sendmail は、ローカルメールの保存方法や配送方法に関しては一切関知しなかった。つまり、単に他のプログラムとの間でメールを入れ替える以外のことは何もしなかったのだ (これは、後に SMTP が追加されたことで変わった。詳しくは後ほど)。ある意味では、自分自身がメールシステムであるというよりは、さまざまなメールシステムの間をつなぐ糊に過ぎなかったのだ。

Sendmail の開発を進めるうちに、Arpanet はインターネットに姿を変えた。その変更点は幅広く、ローレベルのパケットからその上にのるアプリケーションプロトコルまですべて変化した。そして、その変化は一斉に起こったわけではなかった。Sendmail の開発は標準規格の策定とまさに文字通り並行して進んでおり、Sendmail が標準規格に影響を及ぼすこともあった。もうひとつ特筆すべき点は、いま我々が思い浮かべる “ネットワーク” がまだ数百台規模だったころから何億台規模に成長するまでの間、sendmail がずっと生き残って成長を続けたという事実である。

もうひとつのネットワーク

ちなみに、当時まったく別のメール標準規格も提案されていた。それは X.400 と呼ばれ、ISO/OSI (International Standards Organization/Open Systems Interconnect) の一部だった。X.400 はバイナリプロトコルで、メッセージは ASN.1(Abstract Syntax Notation 1) で符号化する。この仕組みは、今でも LDAP などの一部のインターネットプロトコルで使われている。LDAP は X.500 を単純化したものであり、X.500 は X.400 が使っているディレクトリサービスだ。Sendmail は X.400 との直接の互換性に関しては一切考慮していない。しかし、両者の橋渡しをするゲートウェイが存在する。X.400 は当初から多くの商用ベンダーに採用されていたが、最終的に市場を制したのはインターネットメールと SMTP だった。

17.2　設計の原則

Sendmail の開発中にこだわっている設計指針をいくつか紹介する。これらは結局ひとことでまとめることができて、要するに「無駄なことはしない」ということだ。これは、その当時の流れに反するものだった。当時は、より幅広い目標を達成するために実装をどんどん膨らませる方向に進んでいる人が多かった。

プログラマには限界があることを受け入れる

もともと sendmail は、業務としてではなく空き時間を使って書いたものだ。Arpanet メールをカリフォルニア大学バークレー校の人たちがより使いやすくするための手っ取り早い手段として作られた。その肝となるのが既存のネットワーク間でのメールの転送だった。すべての処理はスタンドアロンのプログラムとして実装されており、複数のネットワークが存在することを想定していなかった。既存のソフトウェアにそれなりに手を入れるというのは、一人のプログラマーが空き時間だけでこなすには不可能な作業である。既存のコードをできるだけ変更せずに済み、かつ新しいコードもできるだけ書かずに済むような設計を目指す必

要があった。後述する指針の大半も、この考えに基づいている。仮に大規模なチームがいたとしても、この指針は間違いではなかっただろう。

ユーザーエージェントの再設計はしない

Mail User Agent (MUA) とは、エンドユーザーの多くが “メールソフト” と聞いて真っ先に思い浮かべるもの—メールの読み書きや返信のために使うプログラムである。Mail Transfer Agent (MTA) はそれとはまったく別で、電子メールを送信者から受信者に向けて配送する役割を果たす。Sendmail が書かれた当時、多くの実装はこれらふたつの機能を組み合わせたものであり、協力して開発を進めていた。両方同時に作業を進めるのはキツかったので、Sendmail ではユーザーインターフェイスの問題を完全に投げ捨てた。MUA に対して行った唯一の変更は、自前のルーティング処理の代わりに sendmail を起動させるようにする変更だった。実際、ユーザーエージェントは既にいろいろなものがあったし、実際にメールを操作する部分なのでユーザーの好みも人それぞれだ。MUA を MTA から切り離すという判断は今でこそ受け入れられるだろうが、当時の常識からは完全にかけ離れていた。

ローカルメールストアの再設計はしない

ローカルメールストア (受信者がメールを読むまでの受信メールの保存場所) は、正式には標準化されていなかった。`/usr/mail` や`/var/mail` あるいは`/var/spool/mail` のような場所で中央管理する方式をとっているところもあれば、受信者のホームディレクトリに (`.mail` などといった名前のファイルで) 保存しているところもあった。大半のサイトでは “From” で始まってその後スペースが続く行があればそこからメッセージが始まるものとしていた (あまりにも筋が悪すぎるが、当時はそういう決まりだった) が、Arpanet に注力していたサイトでは、control-A が 4 文字続く行でメッセージを区切って保管していた。一部のサイトではメールボックスをロックして衝突を回避していたが、サイトによってロックの規約が異なっていた (ファイルロックのプリミティブは当時まだなかった)。要するに、やれることがあるとすれば、ローカルのメールストレージは完全にブラックボックスとして扱うことくらいだった。

ほぼすべてのサイトで、実際にローカルメールボックスのストレージを扱う処理は`/bin/mail` に埋め込まれていた。このプログラムは、(極めて原始的な) ユーザーインターフェイスとルーティング、そしてストレージ操作の処理をひとつにまとめたものである。sendmail と組み合わせるために、ルーティング部分は外に切り出して sendmail を呼び出す処理に切り替えた。最終的な配送を強制するための`-d` フラグが追加された。つまり、`/bin/mail` が sendmail を呼び出してルーティングするのをやめさせるということだ。後になって、物理的なメールボックスにメッセージを配送するコードが切り出され、`mail.local` という別のプログラムになった。現在の`/bin/mail` は、メールを送信するための最小限の共通処理だけを残したものになっている。

Sendmail が世界に合わせるのであって、世界を **Sendmail** に合わせるのではない

UUCP や BerkNET といったプロトコルが既に個別のプログラムとして実装されており、それぞれ自前の (時にちょっと奇妙な) コマンドライン構造をもっていた。時には、それらのプログラムも sendmail と同時に活発な開発が進むこともあった。sendmail で再実装してしまう (たとえば、標準の呼び出し規約に変換する) のは、どう見てもつらいだろう。ここから、次の原則が得られる。つまり、sendmail のほうが他の世界に合わせるのであって、他の世界のほうを sendmail の流儀に合わせようとしてはいけないということだ。

変更はできる限り少なく

最大限の注意を払い、sendmail の開発中には、触る必要のないものには一切触らないようにした。単に時間がないというだけの理由ではない。当時のバークレーには、コードの所有者を厳密に定めるのではなく “最後にコードを触った人が、そのプログラムの担当になる”(簡単に言うと “触ったら、責任を持て”) という文化があったのだ。今どきの常識で考えるとあり得ない話だが、これでうまく回っていた。当時のバークレーにはフルタイムで Unix に関わっている人はいなかったのだ。各人が、それぞれ自分が興味を持った部分の作業をしてコミットし、それ以外の部分についてはよっぽどの緊急事態でない限り手を触れなかった。

信頼性をまず考える

sendmail 以前のメールシステム (大半のメール配送システムも含む) は、信頼性についてあまりにも無神経すぎた。たとえば、4.2BSD よりも前のバージョンの Unix にはネイティブなファイルロックの仕組みがなかった。その代わりの手段として、テンポラリファイルを使ってファイルロックをシミュレートしていた。つまり、テンポラリファイルを作成してそれをロックファイルにリンクさせたのだ (もしロックファイルが既に存在すれば、リンクのコールが失敗する)。しかし、時には別のプログラムが、ロックファイルとしての扱い方を知らないまま同じファイルに書き込みをしてしまうこともあり (たとえばロックファイル名が違ったり、そもそもロックを考慮していなかったりなど)、メールを失う可能性もあり得るようになってしまう。sendmail では、メールを失うことなどあり得ないような手法を採用した (これは、もともと私がデータベース屋だったことも関係するかもしれない。データを失うなんてとんでもないという考えだった)。

その他

初期のバージョンでは、まだできていないことがたくさんあった。私はメールシステムを一から作り直そうとはしなかったし、完璧なソリューションを構築しようとも思わなかった。

単に、必要になったときにその機能を追加するようにしたのだ。最初期のバージョンでは、設定を変更したければソースコードをいじってコンパイルし直せといった具合だった (さすがにそれはすぐになくなったが)。sendmail のやり方は、だいたいこんな感じだ。まず、何か動くものを手早く作る。そして、必要に応じて機能拡張し、問題があれば対応する。

17.3　開発フェーズ

古くからあるソフトウェアはだいたいそうだが、sendmail の開発もいくつかのフェーズに分けることができ、個々のフェーズについて基本テーマや考えがある。

第一波: delivermail

sendmail が最初に登場したときには、delivermail と呼ばれていた。単純化しすぎとは言わないまでも、シンプル極まりないものだった。唯一の機能は、あるプログラムから別のプログラムにメールを転送することだった。実際のところ SMTP すらサポートしておらず、直接のネットワーク接続も一切行わなかった。キューなんか不要だった。だって、それぞれのネットワークが自分でキューを持っているんだから。プログラムは、単にクロスバースイッチになればよいだけのことだ。delivermail はネットワークプロトコルを直接にはサポートしていなかったので、デーモンとして動かす理由などなかった。メッセージが投稿されるたびに起動してメッセージを仕分け、次のホップを実装する適切なプログラムにそれを渡し、あとは終了するだけでよい。また、ヘッダを書き換えてメッセージの配送先のネットワークにマッチさせるということもしなかった。そのせいで、転送したメッセージに返信することができないということになりがちだった。さすがにそれはまずかったので、メールの処理だけを扱う本が書かれたりした (*!%:: A Directory of Electronic Mail Addressing & Networks* [AF94] という、ぴったりな名前がついている)。

delivermail はすべてコンパイルされる。また、すべてアドレス内に埋め込まれた特殊文字を利用していた。特殊文字には優先順位があった。たとえば、ホストの設定に関しては “@” 記号を探し、もし見つかれば、そのアドレス全体を割り当てられた Arpanet リレーホストに送信する。見つからなければ次にコロンを探し、割り当てられたホストとユーザー (見つかれば) に BerkNET でメッセージを送る。そして次に感嘆符 (“!”) を探す。これは、そのメッセージを UUCP リレーに転送することを表す。見つからなければ、ローカルへの配送を試みる。この設定をまとめると、次のようになる。
アドレスの区切り文字はその組み合わせによって異なり、結果的に曖昧な状態になってしまって経験則でしか解決できなくなることもある。たとえば最後の例は、別のサイトなら{Uucp, foo, bar@baz}と解釈されることもあり得るだろう。

設定をコンパイルする理由はいくつかある。まず、アドレス空間が 16 ビットでメモリも限られているので、実行時に設定をパースするのはコストがかかりすぎだった。次に、当時の

入力	送信先 ネットワーク、ホスト、ユーザー
foo@bar	{Arpanet, bar, foo}
foo:bar	{Berknet, foo, bar}
foo!bar!baz	{Uucp, foo, bar!baz}
foo!bar@baz	{Arpanet, baz, foo!bar}

システムは個々のサイトで非常にカスタマイズされていたので、再コンパイルをするのも理にかなっていた。ローカルに、必要なバージョンのライブラリがあることを確実にするためにである (共有ライブラリは、Unix 6th Edition の時点ではまだなかった)。

Delivermail は 4.0 BSD および 4.1 BSD に同梱され、予想以上の好評を得た。ハイブリッドなネットワークアーキテクチャを採用しているのはバークレーだけではなかったというわけだ。その証拠に、いろいろな要求が出てきた。

第二波: sendmail 3、4、そして 5

バージョン 1 とバージョン 2 は、delivermail という名前で公開された。1981 年 3 月にバージョン 3 の開発が始まる。このバージョンからは、sendmail という名前で公開されるようになった。この時点ではまだ 16 ビットの PDP-11 が一般的に使われていたが、32 ビットの VAX-11 もポピュラーになりつつあった。当初の制約の中でもアドレス空間による制約については、これで和らぎはじめた。

sendmail の初期の目標は、実行時の設定に変換して別のネットワークへのメッセージの転送をできるよう容易することだった。また、ルーティングを設定するためのリッチな言語を提供することだった。利用していたテクニックは基本的にテキストレベルでのアクセスの書き換え (文字ではなくトークンにもとづいたもの) で、当時の大規模システムでも一般的に使われていたものだ。アドホックなコードを使い、コメント文字列 (括弧で囲まれたもの) を切り出して保存した後でプログラムによる書き換えを終えてからもう一度挿入する。ヘッダフィールドを追加したり拡張したり (たとえば `Date` ヘッダフィールドを追加したり、送信者のフルネームがわかっている場合は `From` ヘッダにそれを含めるなど) できるようにしておくことも大切だ。

SMTP の開発が始まったのは 1981 年 11 月のことだった。カリフォルニア大学バークレー校の The Computer Science Research Group (CSRG) が DARPA と契約を結び、Unix ベースのプラットフォーム作って DARPA が出資する研究をサポートした。その意図は、プロジェクト間の共有をしやすくすることだった。TCP/IP スタックを初めて手がけたのがちょうどこの頃だが、ソケットのインターフェイスの詳細はまだ確定していなかった。Telnet や FTP といった基本的なアプリケーションプロトコルはできていたが、SMTP はまだ実装されていなかったのだ。実際、その時点ではまだ SMTP プロトコルの策定に決着がついていなかった。メールをどのように送信すべきかの議論は収束しておらず、プロトコルの名前は Mail Transfer

Protocol (MTP) と呼ばれていた。議論は白熱し、MTP はどんどん複雑になっていった。結局、業を煮やして SMTP (Simple Mail Transfer Protocol) の提案が出された。多かれ少なかれ、恣意が入っていた (公式に公開されるのは 1982 年 8 月のことだ)。公式には、私がかかわっているのは INGRES Relational Database Management System だった。しかし、当時のバークレーの中で私以上にメールシステムを知っている人はいなかったので、SMTP の実装にもかかわることになった。

私が最初に考えたのは、SMTP を話すメーラーを別に作って自前のキューを持たせ、デーモンとして動かすことだった。このサブシステムを sendmail にアタッチして、ルーティングを任せる。しかし、SMTP の機能のいくつかのせいで、この案は困難になった。たとえば、EXPN コマンドや VRFY コマンドは、パース処理やエイリアス処理そしてローカルアドレスの検証モジュールにアクセスできなければならない。また、当時の私が重視していたのが、RCPT コマンドは未知のアドレスを受け取ったときにすぐに結果を返すということだった。いったんメッセージを受け付けてから後で配送失敗のメッセージを返すのではいけないと考えたのだ。これは後に、先見の明があったと判明する。皮肉なことに、後の MTA の多くはここを誤り、spam の氾濫を招いてしまっている。これらの問題があったので、SMTP は sendmail 本体に含めることに決めた。

Sendmail 3 は 4.1a BSD および 4.1c BSD (ベータ版) に同梱され、sendmail 4 は 4.2 BSD に含まれた。そして sendmail 5 は 4.3 BSD に含まれることになった。

第三波: カオスな日々

バークレーを去ってスタートアップ企業に転職してから、私が sendmail に割ける時間は大幅に減少した。しかし、インターネットの世界はどんどん拡大し続け、sendmail もさまざまな新しい (そして巨大な) 環境で使われるようになった。主要 Unix ベンダーのほとんど (Sun、DEC、そして IBM など) は自前の sendmail を用意しており、お互いに互換性がなくなっていた。オープンソースの sendmail を作ろうという試みもあった。特筆すべきは IDA sendmail と KJS だ。

IDA sendmail は、リンシェーピン大学によるものだ。IDA が含めた拡張は、大規模な環境やまったく新しい構成のシステムへのインストールや管理を容易にするものだった。主な新機能のひとつは dbm(3) データベースを含めたことで、これで活発なサイトもサポートできるようになった。設定ファイルに新しい構文を導入しており、外部のシステムでのアドレスの構文とのマッピング (たとえば johnd@example.com のかわりに john_doe@example.com 宛てにメールを送るなど) やルーティングなどができるようになった。

King James Sendmail (略して KJS。ポール・ヴィクシーが作ったもの) は、そこら中にわき出したさまざまなバージョンの sendmail を統一しようとした試みである。残念ながら、期待していたほどの影響を及ぼすにはとうてい及ばなかった。新しい技術をメールシステムに取り込もうとしすぎて失敗したのだ。たとえば、Sun が作ったディスクレスクラスタに、YP(後の NIS) ディレクトリサービスや NFS(Network File System) を追加したりした。特に、YP は

sendmail に見えなければならない。エイリアスをローカルファイルではなく YP に保存していたからである。

第四波: sendmail 8

数年を経て、私はふたたびスタッフとしてバークレーに戻ってきた。当時の業務は、計算機科学部の研究用共有基盤のインストールやサポートを管理することだった。そのためには、個々の研究グループでアドホックに構築した大規模な環境を何らかのきちんとした方法で統合する必要があった。インターネットの黎明期と同様、研究グループごとにまったく違うプラットフォームを使っており、中には古すぎるものもあった。一般に、すべての研究グループは独自のシステムを持っていた。しかしほとんどはうまく管理されておらず、大半が “繰越維持費” に悩まされていた。

たいていの場合、電子メールはみな整っていない。各個人の電子メールアドレスは “person@host.berkeley.edu” のようになっており、host はオフィスのワークステーション名あるいは共有サーバー名 (キャンパス内では内部サブドメインは使っていなかった) だった。例外として、一部の人は@berkeley.edu アドレスを持っていた。目標は、内部サブドメインを使うようにする (つまり、すべてのホストを cs.berkeley.edu サブドメインに置く) ことと、統一されたメールシステムを使う (つまり、すべての人が@cs.berkeley.edu なアドレスを持つ) ことだった。この目標を実現するための最も簡単な方法は、新しいバージョンの sendmail を学部全体で使わせることだった。

まずは、多数出回っている sendmail の類似品の中でもよく使われているものについて調べることにした。違いを探すというよりはむしろ、他の類似品にはない便利な機能について理解することを心がけた。その過程で見つかったアイデアを元にして sendmail 8 を作っていった。関連するアイデアをひとまとめにしたり、より汎用的にしたりなどの変更を加えることもよくあった。たとえば、sendmail の類似品の中には dbm(3) や NIS のような外部のデータベースにアクセスできる機能を持つものもあった。sendmail 8 では、これらをひとまとめにした “マップ” という仕組みを導入し、複数の形式のデータベース (データベース以外にもどんな変換方式も使える) を処理できるようにしている。同様に、“汎用” データベース (外部の名前マッピングを利用する) 機能は、IDA の sendmail から導入した。

Sendmail 8 には新たな設定パッケージも導入された。これは m4(1) マクロプロセッサを利用したものだ。sendmail 5 の設定パッケージよりもさらに宣言的に記述できるよう心がけ、大部分は手続き型で書けるようになっている。sendmail 5 の場合は、管理者が設定ファイルをすべて手で書き換える必要があった。単に m4 からのショートカットとして “include” を使うだけであってもだ。sendmail 8 の設定ファイルの場合、管理者が宣言できるのは、必要な機能やメーラーなどだけで、最終的な設定ファイルは m4 が作成する。

17.7 節で、sendmail 8 での機能追加について解説する。

第五波: ビジネスの日々

インターネットが成長するにつれて sendmail を使うサイトも増加し、大量のユーザーをサポートするのが徐々につらくなってきた。しばらくの間は、ボランティアの集まり (非公式にだが、"Sendmail Consortium" あるいは sendmail.org と呼ばれていた) のおかげで無料サポートを続けることができていた。サポートには電子メールやニューズグループを使っていた。しかし 1990 年代後半には、インストール数が増えすぎて、もはやボランティアベースでのサポートを続けるのはほぼ無理な状態になっていた。そこで、よりビジネスよりな友人とともに Sendmail, Inc.[2] を設立し、コードの面倒を見る新たな人材を確保しようとした。

そこで扱っていた商品は、設立当初は設定管理ツールが主だった。しかし、オープンソースの MTA にも、ビジネスの世界の要望に対応するためにさまざまな新機能が追加された。特筆すべき点としては、TLS (connection encryption) や SMTP Authentication のサポート、Denial of Service 対策などサイトのセキュリティ対策の向上、そして最も重要なものとしてはメールのフィルタリング用プラグイン (後述する Milter インターフェイス) などがある。

本章の執筆時点では、扱う商品はさらに幅広くなり、電子メールベースの大規模なアプリケーションスイートも含まれるようになっている。そのほぼすべては sendmail の拡張機能として作られており、創業当初の数年間で作られたものだ。

sendmail 6 や 7 はどこに行ったの?

Sendmail 6 は、本質的には sendmail 8 のベータ版だった。公式にはリリースされなかったが、かなり広範囲に広まった。Sendmail 7 が存在することはなかった。バージョン 6 からバージョン 8 に一気に上げたのだ。というのも、1993 年 6 月に 4.4 BSD がリリースされたことにあわせて、BSD ディストリビューションのファイルをすべてそれにあわせたからである。

17.4 設計に関する決定

正しい判断ができたこともあった。当時は正しい判断だったが、その後状況が変わってまずい判断になったものもあった。そして、どちらとも言えない曖昧なものもあった。

設定ファイルの構文

設定ファイルの構文は、次のふたつの要因によって決まった。ひとつは、アプリケーション全体を 16 ビットアドレス空間におさめるために、パーサを小さくする必要があるということ。もうひとつは、初期の設定はごく少なかった (1 ページにおさまる程度) ので、多少構文をあいまいにしても、ファイルがそんなに読みにくくはならなかったということ。しかし、

[2] `http://www.sendmail.com`

時を経て、より多くの判断がCのコードから設定ファイルに追い出されるようになり、設定ファイルが肥大化し始めた。そして、設定ファイルは「難解なもの」という悪評が広まった。多くの人にとってフラストレーションの元となったのが、タブ文字を構文的に意味のある要素とした判断だ。これは、当時他のツール (make など) で使われていた構文をそのまま使っただけだが、間違いだった。この問題が深刻化したのは、ウィンドウシステムが登場したころ (つまり、カットアンドペーストが多用されるようになったころ。カットアンドペーストではタブ文字の情報が残らない) からである。

今思えば、設定ファイルが巨大化して世の中が32ビットマシンに取って代わられた頃に、設定ファイルの構文を検討しなおしてもよかった。そのように考えた時期もあったのだが、結局そうしなかった。当時すでに "大量に" インストールされていた sendmail 環境を壊してしまいたくなかったからである (実際のところ、その時点で実際にインストールされていたマシンは、おそらく数百台程度だっただろう)。この判断は間違っていた。当時の私は、その後インストール数がどれほど伸びるかを想像できなかったし、早めに構文を変更しておくことで今後どれだけの時間を節約できるのかにも思いが及ばなかったのだ。また、標準規格がある程度安定してきた時点で、一般的な項目はもう一度Cのコード側に戻せば、設定ファイルはもう少しシンプルにできただろう。

当時特に気にしていたのは、どれだけの機能を設定ファイルに追い出せるかということだった。私が sendmail の開発を進めていたのは、ちょうど SMTP の標準規格を定めようとしつつある頃だった。SMTP 側で設計の変更があれば、すぐに—通常は24時間以内に—それを設定ファイルに追い出すようにしていた。これは、SMTP の策定にも貢献したと思っている。何か設計の変更が提案されればそれをすぐに実際に試すことができたし、試すためには (難解な) 設定ファイルを書くだけで済んだからだ。

ルールの書き換え

sendmail を書く際に決めづらかったのが、ネットワーク越しの転送を許可するために必要な書き換えを、受信側のネットワークの規約に違反しない方法で行うためにどうすればよいかということだった。ネットワーク越しの通信では、メタ文字の変更 (たとえば BerkNET ではコロンを区切り文字に使っていたが、コロンは SMTP のアドレスには使えない) やアドレスコンポーネントの並べ替え、そしてコンポーネントの追加や削除などが必要となる。たとえば、状況に応じてこのような書き換えが必要となった。

From	To
a:foo	a.foo@berkeley.edu
a!b!c	b!c@a.uucp
<@a.net,@b.org:user@c.com>	<@b.org:user@c.com>

正規表現はあまり良い選択ではなかった。というのも、正規表現ではワードの区切りや

クォートなどにうまく対応できなかったからである。すぐに明らかになったことだが、これに対応する正規表現を書くのは事実上ほぼ不可能で、とてもわかりにくいものになってしまった。正規表現では、いくつかのメタ文字を予約語として使っている。たとえば “.” や “*”、“+”、“[” そして “]” がそれにあたるのだが、これらはみな電子メールアドレスの中に登場しうる文字である。設定ファイルでこれらの文字をエスケープしてしまってもよかったのだが、それは複雑で混乱の元になるし、ちょっと見苦しいと思った (ベル研の UPAS がこの方式を採用していた。これは Unix Eighth Edition のメーラーとして採用されたが、まったくヒットしなかった [3])。そのかわりにスキャンフェイズが必要となり、そこでトークンを切り出して正規表現のように文字を操作することにした。“オペレータ文字” を表すひとつのパラメータをトークンやトークンの区切りとみなせば十分だった。空白文字はトークンを区切るが、それ自身はトークンにはならない。書き換えルールは単なるパターンマッチと置換の組み合わせであり、原則的にサブルーチンとして組み込まれた。

大量のメタ文字をエスケープしてそれらの “特殊” 機能 (正規表現で使われているもの) を消し去るかわりに、私は単一の “エスケープ” 文字を使うことにした。これを通常の文字と組み合わせて、ワイルドカードパターン (任意の単語にマッチする、など) を表すのだ。伝統的な Unix のアプローチなら、ここでバックスラッシュを使うところだろう。しかし、バックスラッシュは既に一部のアドレス構文でクォート文字として使われていた。いろいろ調べた結果見つかった数少ない候補のひとつが “$” で、これなら電子メールの構文で特殊文字としては使われていなかった。

当初の方針の中で間違っていたと思えるのは、皮肉にも、空白の使い方だった。空白は区切り文字で、これはスキャン対象の入力の大半と同様だった。そこで、パターン内のトークンの間で自由に空白を使うことができた。しかし、元々配布されていた設定ファイルには空白は含まれておらず、結果としてパターンが必要以上に読みづらいものとなってしまった。次のふたつのパターンを比較してみよう (文法的にはどちらも同じ意味だ)。

```
$+ + $* @ $+ . $={mydomain}
$++$*@$+.$={mydomain}
```

書き換えを使ったパース

sendmail では文法学にもとづいてアドレスをパースすべきであり、書き換えルールを使うべきではないという提案もあった。書き換えルールを使うのはあくまでもアドレスの書き換えだけにすべきだ、と。聞く限りでは理にかなっているように見える。ただし標準規格でのアドレスの定義が文法学に乗っ取っていればの話だが。書き換えルールを使い回している主な理由は、場合によってはヘッダフィールドのアドレスをパースする必要があるということだ (正式なエンベロープを持たないネットワークから受信したメールで、ヘッダから送信者エンベロープを取り出す場合など)。この手のアドレスのパースは、YACC のような LALR(1) パーサや伝統的なスキャナでは困難である。というのも、相当な先読みを要求されるからだ。た

[3] http://doc.cat-v.org/bell_labs/upas_mail_system/upas.pdf

とえば、allman@foo.bar.baz.com <eric@example.com>のようなアドレスをパースすることを考えると、スキャナあるいはパーサで先読みが必要となる。つまり、最初の “allman@…” がアドレスではないということは、少なくとも “<” まで読まなければわからない。LALR(1) パーサには先読みトークンがひとつしかないので、これはスキャナで行う必要がある。相当複雑な作業だ。書き換えルールには既にいくらでも後戻りできる (つまり、いくらでも先読みできる) ので、こちらのほうが適している。

第二の理由は、パターンの認識がしやすいうえに壊れた入力も修正しやすかったということだ。最後の理由は、やりたいことをこなすには書き換えでも十分すぎるくらい高機能だったからである。それに、コードを再利用できるならそうするほうが賢いというものだ。

書き換えルールに関して特筆すべき点がひとつある。パターンマッチを行うときには入力とパターンをともにトークンに分割したほうが有用だ。そうすれば、入力アドレスとパターンそのものに対して同じスキャナを使える。そのためには、スキャナを呼び出す際に、入力ごとに文字タイプテーブルを切り替えられるようにしなければならない。

SMTP やキューの sendmail への埋め込み

SMTP の送出側 (クライアント側) の実装として “自明な” 方法は、UUTP と同様に外部のメーラーとして実装することだ。しかしこの場合、いくつか疑問が出てくる。たとえば、キューイングは sendmail でするのだろうか?それとも SMTP クライアントモジュールで行うのだろうか?sendmail で行うのなら、メッセージのコピーを各受信者に送信する (つまり、“piggybacking” はせず、そこで一つの接続を開き、複数の RCPT コマンドを送る) か、あるいはずっとリッチな逆方向のコミュニケーション手段が必要になる。受信者ごとの状況を知るには、単純に Unix の終了コードを使うだけでは不十分だからである。クライアントモジュール側で行うのなら、大量の複製が発生する可能性が出てくる。特に、当時は XNS など他のネットワークもまだ候補にあった。さらに、キューを sendmail 側に含めれば、よりエレガントな方法で障害に対応できるようになる。特に、リソースの枯渇などの一時的な問題に対応しやすい。

受け入れ側 (サーバー側) の SMTP については、難しい決断があった。当時の私は、VRFY や EXPN といった SMTP コマンドも忠実に実装することを重視していた。これらのコマンドは、エイリアスの仕組みにアクセスできなければならない。これを実現するには、SMTP サーバーモジュールと sendmail との間でよりリッチなプロトコルの交換が必要となる。これは、単にコマンドラインと終了コードだけで実現できるものではない—実際のところ、SMTP 自体はそのようなプロトコルだったのだが。

今なら、キューイングは sendmail のコアに残すだろうが SMTP のクライアント側、サーバー側の実装はそれぞれ別のプロセスに切り出してしまいたいところだ。その理由のひとつはセキュリティの向上である。サーバー側でいったん 25 番ポートを開いたインスタンスを持てば、もはや root 権限は不要になる。TLS や DKIM 署名のようなモダンな拡張のせいでクライアント側は複雑になるが (権限を持たないユーザーに秘密鍵へのアクセスを許してはいけないから)、厳密に言えばこちらも root 権限は不要だ。しかし、セキュリティの問題は依然

問題として残る。クライアント側の SMTP が非 root ユーザーで稼働していたとしても、秘密鍵を読めるということは特別な権限を持っていることになる。つまり、他のサイトと直接通信すべきではないということだ。これらの問題はすべて、多少の手間でなんとかできる。

キューの実装

sendmail は、当時の規約に従ってキューファイルを格納していた。実際のところ、採用したフォーマットは当時の lpr サブシステムと極めて似たものだった。各ジョブに対してふたつのファイルがあり、ひとつが制御情報でもうひとつが実際のデータとなっていた。制御ファイルはフラットなテキストファイルで、各行の最初の文字がその行の意味を表していた。

sendmail がキューを処理するときは、制御ファイルをすべて読んだ上で関連する情報をメモリ内に格納し、そしてリストをソートしなければならない。キュー内のメッセージ数が比較的少ないうちは、これはうまく機能した。しかしキュー内のメッセージが 10,000 前後になった時点から不調になり始めた。特に、ディレクトリが肥大化してファイルシステム内の間接ブロックを要するようになると、そこがパフォーマンスに深刻な影響を及ぼす。大幅にパフォーマンスが落ちてしまうこともあり得る。この問題を改善する手段として sendmail で複数のキューディレクトリを扱えるようにすることもできた。しかしそうしたところで、せいぜいちょっとしたハック程度の効果しか得られないだろう。

別の実装としては、すべての制御ファイルをひとつのデータベースファイルにまとめることもできただろう。そうしなかった理由は、sendmail を書き始めた頃にはまだ汎用的に使えるデータベースパッケージが存在しなかったことだ。後に登場した dbm(3) にはいくつかの問題があった。たとえば、領域の再配置 (すべてのキーを 512 バイトの単一ページにまとめるために必要) ができないことやロックの仕組みがないことだ。堅牢なデータベースパッケージは、なかなか登場しなかった。

それ以外にも、別のデーモンを用意してキューの状態をメモリ上に保持させるという方法もあっただろう。そのデーモンがログを書いておけば、リカバリも可能だ。当時はまだ電子メールのトラフィックがそれほど多くなかったこと、そしてメモリを潤沢に搭載したマシンが少なかったこと、バックグラウンドプロセスのコストが比較的高いこと、プロセスの実装が複雑になることなどを考慮すると、当時としてはこの選択肢は割に合わなかった。

もうひとつの設計上の判断として、メッセージヘッダをキューのデータファイルではなく制御ファイル側に格納するようにした。その根拠は、ほとんどのヘッダはそれなりの書き換えを要し、書き換え方法も配送先によってさまざまに変わるということ (そしてメッセージの配送先が複数になることもあり、複数回のカスタマイズが必要になる)。そしてヘッダのパースのコストが高そうだったことだ。そこで、パース済みの形式でヘッダを格納しておけばコストを抑えられるだろうと考えた。今思えば、これはあまりよい判断ではなかった。ちょうどメッセージ本文の格納に Unix 標準フォーマット (行末が newline) を使い、受信したメッセージのフォーマット (行末は newline かもしれないし carriage-return/line-feed かもしれない。単に carriage-return だけかもしれないし line-feed/carriage-return かもしれない) を使わなかった

ように。電子メールの世界が成長して標準が定まっていくにつれて、書き換えの必要性は少なくなった。さらに、どうってことないように見える書き換えであってもエラーのリスクがある。

誤入力の受け入れと修正

sendmail が作られた当時の世界にはさまざまなプロトコルが乱立しており、標準規格もほとんど定まっていなかった。そのため、不正な形式のメッセージはできる限りきれいにしようと決めた。これは、RFC 793 に明記された “ロバストネス原則” (またの名をポステルの法則)[4] にもマッチしている。これらの変更の中には、明白であり必須なものもある。UUCP メッセージを Arpanet に送信するときには、UUCP アドレスを Arpanet アドレスに変換しないと単なる “reply” コマンドすら正常に動作しない。また、行末文字をさまざまなプラットフォームの決まりにあわせて変換したりなどといった作業も必要だ。中には自明だとまでは言えないものもある。受信したメッセージに `From:` ヘッダーフィールドがない場合はどうすればいいだろう?このフィールドは Internet の仕様では必須のものだが、ここで `From:` ヘッダーフィールドを追加してしまうべきだろうか?それとも `From:` ヘッダーフィールドを追加せずにそのまま通してしまうべきだろうか?あるいはそのメッセージを拒否すべきだろうか?当時の私が最重要視していたのは相互運用性だった。そこで sendmail では、メッセージにパッチをあてて `From:` ヘッダーフィールドを追加するなどしていた。しかしそのせいで、壊れている他のメールシステムがそのままの状態で長く生き続けることになってしまった。本来なら、ずっと前に修正されるなりこの世から消えてしまうなりすべきだったはずなのに。

私の判断は、その当時は正しかったと確信している。しかし今となっては問題もある。高度な相互運用性を維持することは、メールの流れを妨げないためにも重要だった。もし不正なメッセージを拒否していたら、当時のメッセージの大半は拒否されてしまっていただろう。もし何も手を入れずに素通ししていたら、受け取ったメッセージに返信できなくなってしまう。というか、そもそも誰がそのメッセージを送ったのかさえわからなくなる—あるいはそのメッセージが別のメーラーに拒否されたのかどうかもわからない。

現在は標準規格がきちんと定まっており、そのほとんどの部分は正確で完全になっている。もはや、大半のメッセージが拒否されてしまうという状況ではなくなった。しかし今もなお、不正な形式のメッセージを送るメールソフトが残っている。そのせいで、インターネット上の他のソフトウェアの間でも無駄に多くの問題が発生している。

設定ファイルでの M4 の使用

ある時期、sendmail の設定ファイルに頻繁に変更を加えて個人的にさまざまなマシンに対応させようとしていたことがあった。設定ファイルの大半はマシンが異なっても同じだったた

[4] “自分には保守的であれ。他者はリベラルに受け入れよ。”

め、できれば何かツールを使って設定ファイルを作れたらいいなと考えていた。m4 マクロプロセッサは Unix に含まれるツールである。もともとは、プログラミング言語 (特に RATFOR) のフロントエンドとして設計された。最も重要なのは、m4 が “include” 機能に対応していたことだ。これは C 言語における “#include” と同様の機能である。最初の設定ファイルはこの機能に毛が生えた程度で、ちょっとしたマクロによる拡張をしただけのものであった。

IDA sendmail も m4 を使っていたが、その使い方はまったく異なっていた。今思えば、当時の私はこれらのプロトタイプをもっと調査するべきだったのだろう。彼らはいろいろうまい使い方をしており、特にクォートの処理方法がすばらしかった。

sendmail 6 以降、m4 設定ファイルは新たに書き直され、より宣言的なスタイルで分量も少なくなった。この設定ファイルは m4 プロセッサのパワーをさらに利用するものとなったが、そのために、GNU m4 がその構文を微妙に変更しただけでも問題になることがあった。

最初に考えていたのは、m4 の設定は 80/20 ルールに従って使おうということだった。つまり、m4 を使うのは作業全体の 20%だけにしておけばファイルがシンプルになるし、その 20%が問題の 80%をカバーしてくれるということだ。この計画はすぐに頓挫した。その理由はふたつ。まずは些細な理由のほうから話す。少なくとも最初のうちはこの計画はうまくいき、問題の大半は比較的楽に処理できていた。しかし sendmail やそれを取り巻く世界が拡大するにつれて、徐々に難しくなってきた。TLS 暗号化や SMTP 認証などの機能が組み込まれても、それに対応するのに時間がかかるようになったのだ。

もうひとつの重要な理由は、生の設定ファイルを直接さわるのがもはやほとんどの人にとって難しくなりつつあったということだ。要するに、生の `.cf` フォーマットはアセンブラのコードに等しい状態になった。細心の注意を払えば編集できるが、現実的な話ではない。そこで、m4 スクリプトで書かれたその “ソースコード” が `.mc` ファイルとして格納されることになった。

もうひとつの重要な特徴は、生の設定ファイルは実際のところプログラミング言語だったということだ。手続き型のコード (ルールセット) やサブルーチンの呼び出し、パラメータの展開、そしてループなどの機能があった (しかし goto は使えない)。その文法は曖昧なものものだったが、おおまかには `sed` や `awk` と似ていた。少なくとも概念的には。m4 フォーマットは宣言型だった。低レベルの言語機能を使うこともできたが、現実的にはこれらの詳細はユーザーからは隠されていた。

m4 を使うという判断が正しかったのかどうかはよくわからない。当時の私が考えていた (そして今もそう思っている) のは、複雑なシステムを便利に使うためには、何らかの DSL(ドメイン特化言語) を実装してシステムを構築できるようにするとよいということだ。しかし、この DSL を設定項目としてエンドユーザーに公開すると、システムの設定がプログラミングの問題になってしまう。DSL は強力なものだが、それを使うためのコストも無視できない。

17.5　その他の検討事項

それ以外にも、アーキテクチャや開発に関して話しておくべきポイントがある。

インターネット全体に広がるシステムの最適化

ネットワークベースのシステムのほとんどには、クライアント側とサーバー側の対立がある。クライアント側にとって都合のいい戦略はサーバー側にとってはあまりよろしくないものであることもあるし、その逆もまたあり得る。たとえば、サーバー側での処理コストを最小限に抑えようとすれば、できる限りの情報をクライアント側にプッシュすることになる。一方クライアント側でも同じように考えたとすると、やることは同じだがその方向が正反対になる。たとえば、サーバー側では spam 処理の間も接続を維持したままにしておきたいだろう。そうすればメッセージの配送を拒否する際のコストが下がるからである (最近は、配送を拒否することが一般的になった)。しかし、クライアント側としてはできるだけ早く次に進みたいはずだ。システム全体を見て、インターネット全体のことを考えれば、クライアント側とサーバー側のニーズに対してうまくバランスをとるのが最適なソリューションとなる。

クライアント側あるいはサーバー側のいずれか一方を明示的に重視した戦略をとる MTA もいくつかあった。そんなことができたのは、それらの MTA があまり広まっていなかったからというだけの理由にすぎない。自分のシステムがインターネット上の大部分で使われるようになると、両者にかかる負荷のバランスを考慮した設計が必要となる。インターネット全体を最適化するためだ。これは複雑で込み入った作業となる。なぜなら、MTA というものは常にどちらか一方向に完全に傾いてしまっているものだからである。たとえば大量メール配信システムはメールの送出側の最適化しか気にしない。

接続の両サイドにかかわるシステムを設計する際には、どちらか一方に注意を向けすぎないようにすることが大切だ。これは、よくあるクライアントとサーバーの不調和とはまったく対照的な話であることに注意しよう—たとえばウェブサーバーとそのクライアントは、通常は別々のグループが開発しているものだ。

Milter

sendmail への機能追加のうち最も重要なもののひとつが milter (*m*ail f*ilter*) インターフェイスだ。milter はオフボードのプラグインとして (つまり、別プロセスで実行させて) メールの処理に使える。もともとはスパム対策のために作られたものだった。milter のプロトコルは、サーバーの SMTP プロトコルと同期して動作する。新たな SMTP コマンドをクライアントから受信するたびに、sendmail は milter を呼び出してそのコマンドからの情報を渡す。milter は、その内容にあわせてコマンドを受け入れたり拒否したりする。拒否した場合は SMTP プロトコルでのそのコマンドの実行が却下される。milter はコールバックとして設計されており、SMTP コマンドを受け取ると適切なサブルーチンが呼び出される。milter はスレッド化されている。接続単位のコンテキストポインタを保持しており、各ルーチンに状態を渡すことができる。

理屈上は、milter は sendmail のアドレス空間内にロードできるモジュールとしても動かせる。しかしそうしなかった。理由は次の三点である。まず第一に、セキュリティの問題が重

大だった。仮に sendmail 専用に root 権限のないアカウントを作って運用していたとしても、そのユーザーはすべてのメッセージの状態にアクセスできるようになる。同様に、milter の作者が sendmail の内部状態にアクセスしようとすることも避けられない。

第二に、我々は sendmail と milter の間にファイアウォールを作りたかった。仮に milter がクラッシュしたとしてもそこで被害を食い止め、メールの流れは妨げられないようにしたかった。第三に、milter の作者にとっては、スタンドアロンのプロセスをデバッグするほうが sendmail 全体を相手にするよりもずっと楽だった。

milter がスパム対策以外にも有用であることに、間もなく気づきだした。実際、milter.org[5] のサイトにはさまざまな milter が掲載されている。スパム対策以外にもウィルス対策やアーカイブ、コンテンツ監視、ログ出力、トラフィックの削減などさまざまなカテゴリがあり、商用製品もあればオープンソースのプロジェクトも存在する。postfix[6] も、同じインターフェイスで milter をサポートするようになった。milter は、sendmail の大成功を示す一例と言える。

リリーススケジュール

よく議論になるのが、“早めに、そして頻繁にリリース” 派と “安定したシステムをリリース” 派の対立である。sendmail は、このどちらの手法も繰り返し使ってきた。かなり大量の変更をするときには、一日に何度もリリースすることもあった。基本的な考えは、何か変更するたびにリリースするというものだった。これは、ソース管理システムのツリーを一般向けに公開するのと同じようなことだ。私は個人的に、頻繁にリリースするほうがソースツリーを公開するよりも好きだ。少なくともその理由の一部になっているのは、私のソース管理システムの使い方があまり一般的ではないということだ。大規模な変更をするときなど、コードを書いている途中できちんと動作しない状態のスナップショットをチェックインすることもある。もしツリーを公開することになったら私はブランチを切ってスナップショットを扱うことになるだろう。しかし、いずれにしてもそれが全世界に晒されてしまうわけで、かなり戸惑わせてしまうことになる。また、リリースをするということはそれに番号を振るということであり、番号をつけておけばバグレポートに対して変更を追いかけやすくなる。もちろんこのやりかたで進めるにはリリース作業が簡単でなければならないが、常にそうだとは限らない。

sendmail がクリティカルな本番環境で使われるようになり始めると、この方式では問題が出てきた。私が行った変更が「みんなにちょっと試してもらいたいもの」なのか「本番環境に適用して欲しいもの」なのか、それが通じないことが出てきたのだ。リリースするときに “alpha” とか “beta” とか名付けておけばいくらかましになるが、それでも問題は解決しない。その結果どうなったかというと、sendmail が成熟するにつれてリリースの頻度が下がってリリースあたりの変更が大きくなった。これが特に問題となったのは、sendmail が営利企業に

[5] http://milter.org
[6] http://postfix.org

組み込まれたときだった。顧客は最新のイケてるバージョンを望むが、同時に安定版も望む。そして、その二つが両立しないという事実を認めようとしないのだ。

この手のオープンソース開発者のニーズと商用製品のニーズとの対立は、決してなくならない。早めに頻繁にリリースすることには多くのメリットがある。特に、勇気のある (時に無謀な) テスターを多く獲得できるのが大きい。彼らによるテストは、標準的な開発環境では決して再現できないだろう。しかし、プロジェクトがうまく進むとそれは徐々に製品に姿を変える傾向がある (オープンソースであろうとフリーであろうと関係ない)。そして製品になってしまうとそのニーズはプロジェクトのニーズとは変わってくる。

17.6　セキュリティ

sendmail のこれまでの生き様は、セキュリティの面では波乱に満ちたものだった。さまざまな波乱の中には、起きてしかるべきものもあればそうでないものもあった。そして我々が考える “セキュリティ” の概念も変わってきた。インターネットが始まった頃のユーザー数はたかだか数千人程度で、その大半は学術研究に関わる人たちだった。古き良き時代。いろんな意味で、今のインターネットよりも親切で紳士的だった。ネットワークの設計は情報の共有を推奨する作りになっていたし、ファイヤウォールを構築するなどという考えはなかった (そんな概念は、そもそも初期のインターネットには存在しなかった)。今やネット上には危険がいっぱい。悪意に満ちた場所となり、スパマーやクラッカーがそこらじゅうにあふれている。インターネットが紛争地帯にたとえられることも増えてきた。紛争地帯には、民間人の犠牲者もつきものだ。

ネットワークサーバーをセキュアに書くのはとても難しい。プロトコルがほんの少しでも複雑になってくると、なおさらだ。どんなプログラムだって、少なくとも些細な問題は抱えている。一般的な TCP/IP の実装でさえ、外部からのアタックを受けてしまった。より上位レベルの実装言語になると、万能薬は存在しない。そしてその言語自身が脆弱性を作ることもある。必ずといっていいほど見るフレーズが、どこから来たものかにかかわらず “すべての入力は疑ってかかれ” というものだ。疑うべき入力には、二次的な入力も含まれる。たとえば DNS サーバーや milter からの入力もそうだ。初期のネットワークソフトウェアの大半がそうだったのだが、sendmail の初期のバージョンでは入力を信頼しすぎていた。

しかし、sendmail で最大の問題だったのは、初期のバージョンが root 権限で動作していたという点だ。root 権限を要した理由は、SMTP をリスンするソケットを開いたり各ユーザーの転送する情報を読んだり、各ユーザーのメールボックスやホームディレクトリにメッセージを配送したりするためだった。しかし、今どきのシステムの大半では、メールボックスの名前とシステム上のユーザーの概念とが切り離されている。そのため、SMTP をリスンするソケットを開く以外の操作では root 権限は事実上不要となった。現在の sendmail では、接続を処理する前に root 権限を放棄できるようになっている。これをサポートしている環境なら、root 権限に関する問題は気にしなくてもよくなった。さらに、ユーザーのメールボック

スへの直接の配送をしないシステム上では、sendmail を chroot 環境で動かすこともできる。そうすれば、さらに権限を隔離できる。

不幸にも、sendmail のセキュリティが貧弱だという評判が広まるにつれて、まったく関係のない問題までも sendmail のせいにされることが出てきた。たとえば、あるシステム管理者は、自分が/etc ディレクトリに書き込み権限を与えておきながら、誰かが/etc/passwd ファイルを書き換えたときにそれを sendmail のせいにしたりした。そうした事件を経て、我々もセキュリティについてより真剣に考えるようになった。そして、sendmail がアクセスするファイルやディレクトリの所有者やモードを、明示的にチェックするようにした。このチェックは非常に厳しいものだったので、DontBlameSendmail オプションを用意してこのチェックを無効化できるようにもした。

それ以外の観点からのセキュリティ問題もあった。プログラム自身のアドレス空間を守るのとは直接関係しないものだ。たとえば、spam が増加するにつれて出てきた、メールアドレス収集に関する問題がこれにあたる。SMTP の VRFY コマンドや EXPN コマンドはそれぞれ、個別のアドレスを検証したりメーリングリストのメンバーを展開したりするために用意されたコマンドである。スパマーたちに悪用されることがあまりにも多発したので、ほとんどのサイトでは今やこのコマンドは無効化されている。少なくとも VRFY に関しては、これは残念なことだ。このコマンドは、アンチスパムエージェントが送信者のアドレスを検証するときに使うこともあるからである。

同様に、ウィルス対策の保護もかつてはデスクトップ側の問題とみられていた。しかし、その重要性が増すにつれて、商用レベルの MTA ではウィルス対策のチェックをできることが当然になってきた。その他、最近の設定でセキュリティ関連の必須要件となったものとしては、重要なデータを強制的に暗号化させることやデータの喪失に対する保護、HIPPA などの法的規制への対応などがある。

初期の sendmail が最重要視していたのは信頼性だった。つまり、あらゆるメッセージをきちんと配送する (あるいは送信者に差し戻す) ということだ。しかしジョージョブ問題 (発信元アドレスを偽装したメールによる攻撃。多くの人はセキュリティの問題ととらえている) のせいで、多くのサイトがバウンスメッセージの作成を無効にしてしまった。もし SMTP 接続が開いている間に失敗を検出できれば、サーバー側で SMTP コマンドを失敗させることで何か問題が起こったことを伝えられる。しかし、SMTP 接続が閉じてしまった後だと、アドレスが間違っていたメッセージはただ黙って消えてしまうだけになってしまう。最近のメールの大半は 1 ホップで届くので、何か問題があればわかるだろう。しかし、少なくとも理屈上は、この世界では信頼性よりセキュリティを重視するようになったということだ。

17.7 Sendmail の進化

激動の環境でソフトウェアが生き残り続けるには、環境の変化にあわせて自らを進化させなければならない。新たなハードウェア技術が登場すればそれにあわせて OS も変化するし、

OSが変わればライブラリやフレームワークも変わる。つまりそれはアプリケーションにも変化を促すことになる。アプリケーションが存続すればするほど、問題のある環境で使われることも多くなる。変更は避けられない。生き延びるためには変更を受け入れてそれを取り込まねばならない。このセクションでは、これまでsendmailに起こった変化の中で重要なものをいくつか取り上げる。

設定はより冗長に

当初のsendmailの設定は、極めて簡潔だった。たとえば、オプションやマクロの名前はすべて一文字だった。その理由は次の三つである。まず第一に、そうすればパース処理をシンプルにできるということ (16ビット環境ではこれが重要だった)。第二に、オプションの数がそれほど多くなかったこと。一文字のオプションを考えるのにそれほど苦労しなかった。第三に、単一文字の規約が既にコマンドラインのフラグで確立されていたこと。

同様に、書き換えルールセットも、当初は名前ではなく番号で指定していた。ルールセットの数が少ないうちは、これでもまだ耐えられた。しかし、その数が増えてくると、よりわかりやすい名前をつけることが大切になった。

sendmailの稼働環境が複雑になるにつれ、また同時に16ビット環境が去りゆくにつれ、よりリッチな設定方法の必要性が明らかになってきた。幸いにも、後方互換性を維持したままでの変更が可能だった。この変更によって、設定ファイルのわかりやすさが劇的に向上した。

他のサブシステムとの接続: さらなる統合

sendmailが書かれたころは、メールシステムといえば大抵はOSの他の部分から隔離されているものだった。統合を要するサービスはごく一部で、たとえば`/etc/passwd`や`/etc/hosts`といったファイルくらいだった。サービスを斬り合える機能はまだ発明されていなかったし、ディレクトリサービスも存在しなかった。そして設定ファイルはまだ小さく、手で書けるレベルだった。

状況はすぐに変わった。最初に追加した中のひとつがDNSだ。システムが持つホストルックアップの抽象化機能 (`gethostbyname`) は、IPアドレスを探すには充分だったが、メールでは、それ以外にもMXなどを問い合わせる必要があったのだ。後に、IDA sendmailでは外部データベースを使ったルックアップ機能を組み込んだ。このデータベースにはdbm(3)ファイルを使った。sendmail 8ではそれをさらに拡張し、汎用的なマッピングサービスを用意して他の形式のデータベースも使えるようにした。外部のデータベースも使えるし、内部での変換も使える。内部での変換は、書き換え機能 (アドレスのクォート解除など) なしには実現できなかっただろう。

今やメールシステムは多数の外部サービスに依存しており、一般的に、電子メール専用として設計されることはなくなった。そのため、sendmailのコードもより抽象化する方向に進

んだ。そのおかげで、メールシステムの開発や保守はより難しいものとなった。“可動部品”がいろいろ増えたからである。

ギスギスした世界

sendmail の開発が始まったころの世界は、現在の常識からするとまったく異質の世界に見えることだろう。初期のネットワークにかかわっていた人の大半は研究者で、比較的温和な人が多かった。学術思想についての争いでたちの悪い振る舞いが起こることもあったが、すぐに収まった。sendmail は当時の世情を反映して作られており、メール配送の信頼性を上げることを最重要視していた。たとえユーザー側で間違いがあったとしても、可能な限りメールを配送できるように心がけた。

今の世の中は、当時に比べてはるかにギスギスしている。飛び交う電子メールの大半は、悪意のあるものだ。MTA が目指すゴールは、メールをきちんと配送することから悪意のあるメールを排除することに変わった。いまどきの MTA で最重要視されるのは、きっとフィルタリング機能だろう。それに対応するため、sendmail にも数多くの変更が必要となった。

一例として、数多くのルールセットが追加された。これらを使って SMTP コマンドのパラメータをチェックし、問題を早期に発見できるようになった。エンベロープを読んだ段階でメッセージを拒否するほうが、メッセージ全体を読んでから拒否するよりもはるかにコストが低くなる。ましてや、メッセージの配送を受け入れた後で拒否するようにするとずっと高くついてしまう。初期のフィルタリングは、メッセージをいったん受け入れた後でフィルタプログラムに渡し、フィルタを通過したものだけを別の sendmail インスタンスに送るという仕組みだった (いわゆる “サンドウィッチ” 構成である)。いま同じようなことをすれば、非常にコストがかかることだろう。

また、sendmail での TCP/IP 接続の使い方も変わった。もともとはごく標準的に使っているだけだったが、最近はより洗練されている。ネットワーク入力を “のぞき見” して、前のコマンドが受理されていないのに送信者が次のコマンドを送信していないかどうかを調べたりもする。このため、sendmail が複数のネットワークを扱えるように作られた抽象化の中にはうまく動かないものも出てきた。今でも、sendmail がたとえば XNS や DECnet のネットワークにも接続できるようにかなりの作業をしている。しかし、TCP/IP に固有の知見が多くのコードに取り込まれてしまっている。

ギスギスした世界に立ち向かうために、さまざまな設定項目が追加された。アクセステーブルへの対応やリアルタイムブラックリストへの対応、アドレス収集対策、DoS 対策、そして spam のフィルタリングなどだ。そのおかげでメールシステムの設定はかなり複雑な作業になってしまったが、今の世の中で生きていくためにはそうするしかなかった。

新技術の採用

新たな標準規格が続々と誕生し、それらもまた sendmail に大きく手を加える要因となった。たとえば TLS(暗号化) を追加するときには、大半のコードを変更しなければならなかった。SMTP のパイプラインを実現するには、ローレベルの TCP/IP ストリームを注視してデッドロックを回避する必要があった。サブミッションポート (587) をサポートするには、複数の入力ポートを待ち受ける機能が必要となった。そして、ポートによって挙動を変えなければならなかった。

それ以外にも、標準規格ではなくその場の状況に押されて追加した機能もある。たとえば、milter インターフェイスを追加したのは spam に耐えられなくなったからだ。milter は標準規格として確立されたものではなかったが、大きな新技術だった。

どの場合についても、これらの変更によって何らかの面でメールシステムは強化された。セキュリティの向上やパフォーマンスの向上、あるいは新機能の追加などによってである。しかし、どの場合についてもそれなりのコストがかかっており、ほぼすべての変更がコードや設定ファイルを複雑化させている。

17.8 もし今やりなおせるとしたら?

後からなら何とでも言える。というわけで、今だったら違うやりかたにしただろうということもたくさんある。当時としては予測不能だったこと (spam のせいでメールに対する視点がどれほど変わるかや、最新のツール軍がどんなものになるかなど) もあれば、どう見ても予測できたはずだろうということもある。また、sendmail を書いている間に私自身さまざまなことを学んだ。電子メールのことや TCP/IP のこと、そしてプログラミング自体のことなど。誰もがコードとともに成長するということだ。

しかし、今でも同じようにするだろうということもたくさんあって、その中には一般的な常識に反するものもある。

違うやりかたにしたいところ

おそらく sendmail 史上最大の過ちは、後に自身がどれだけ重要な存在になるのかを早期に気付けなかったことだろう。世界が正しい方向に向かうようにほんの少し突っつける機会が何度かあったはずなのに、そうしなかった。実際、たとえば sendmail の入力チェックを厳しくして不正な入力を拒否するようにもできたはずだが、そうすべきときに実際は何もしなかった。同じく、設定ファイルの構文に改善の余地があることもわかっていた。まだ世界中で数百インスタンス程度しか sendmail が動いていなかったころの話だ。わかってはいたのだが、結局そのときは変更しなかった。その時点での既存ユーザーに負担を強いることになるからである。今思えば、何かを改善するなら早めにしておくべきだった。一時的につらいこともあるが、長い目で見ればよりよい結果になっただろう。

バージョン 7 の **Mailbox** の構文

その一例が、バージョン 7 のメールボックスでのメッセージの分割方法である。当時は "From␣" ("␣" は ASCII の空白文字、つまり 0x20) で始まる行でメッセージを分割していた。もしメッセージ本文の中に "From␣" で始まる行があれば、ローカルのメールボックスソフトウェアはそれを ">From␣" に変換していた。いくつかのシステムでは「その前に空行を含むこと」を要件とできたが、すべてではなかったのでそれに依存はできなかった。今日に至るまで、">From" はありとあらゆる箇所で予期せず登場している。それはメールとは直接関係のない (しかし、いつだったかそれがメールで処理されたことがあった) 場面も含む。今思えば、BSD メールシステムから別の形式の構文に変換したほうがよかった。変更した直後には多くの人から恨まれるだろうが、そうしておけばさまざまな問題からこの世界を救えたはずだ。

設定ファイルの構文とその中身

おそらく、設定ファイルの構文における最大のミスは、書き換えルールの記述でパターンと置換内容の区切りにタブ (HT, 0x09) を採用したことだろう。その当時は、make の流儀をまねたのだった。しかしその数年後に make の作者である Stuart Feldman に聞いた話によると、彼もまたタブを採用したことをいちばん後悔しているとのことだった。設定ファイルを画面上で見たときにタブがあるかどうかがわかりにくいというだけでなく、大半のウィンドウシステムではカットアンドペーストでタブが消えてしまうという問題もあった。

書き換えルールという考え方自体は間違っていなかったと思っている (以下を参照)。しかし、設定ファイルの全体構造はもう少しなんとかできただろう。たとえば、設定ファイルで階層構造が必要になることを想定できていなかった (SMTP リスナーのポートごとに異なるオプションを設定したりなど)。当時、設定ファイルを設計するにあたっては、"標準" フォーマットなど存在しなかった。今なら Apache 形式の設定構文を採用するだろう。すっきりしているし、充分な表現力もある。あるいは、Lua などの組み込み言語を使ったかもしれない。

sendmail の開発が進んでいたころは、アドレス空間も小さいしプロトコルもまだ流動的だった。可能な限り設定ファイル側に押し出しておくのが無難だったのだ。今の状況では、これは間違いだろう。今や MTA は広大なアドレス空間を使えるし、標準規格もきちんと固まっている。さらに、"設定ファイル" の一部は実際のところプログラムのコードのようになってしまっており、新しいリリースが出るたびに更新が必要になるほどだ。`.mc` 設定ファイルを使えばこの問題は解決できるが、ソフトウェアを更新するたびに設定ファイルをビルドしなおす必要があるというのもつらいことだ。この問題に対するシンプルな対策は、sendmail が読む設定ファイルを二つに分割することだろう。一方は利用者から見えないようにしてソフトウェアの更新時に自動的にインストールされるようにし、もう一方を公開してローカル設定用に使わせるという方法だ。

ツールの活用

今ではさまざまな新しいツールが登場している。たとえばソフトウェアの構成やビルドひとつとってもそうである。必要に応じてツールを活用すればその力を生かせるだろうが、時にやり過ぎてしまうこともある。そうなれば、システムを理解するのが必要以上に困難になってしまう。たとえば、単に strtok(3) があれば充分なときにわざわざ yacc(1) の構文を使うのはばかげている。しかし、車輪の再発明をするのもあまりよい考えではない。例を挙げると、私はよっぽどの場合を除いて autoconf を使うようにしている。

後方互換性

もし将来の姿が見えていて、sendmail がいかに普及するかがわかっていたなら、開発初期の段階で既存の環境との互換性を崩してしまうことを躊躇しなかっただろう。もし既存の習慣がうまくいかなくなってしまうのならそれを修正すべきであって、なんとか対応させるというのではいけない。とはいえ、私はまだメッセージフォーマットの厳格なチェックをしていない。単に無視できたり簡単に手直しできたりするような問題もあるからである。たとえば、`Message-Id:`ヘッダーがないメッセージには今でもきっとヘッダーを追加するだろう。しかし、`From:`ヘッダーがないメッセージについては、わざわざエンベロープの情報からヘッダーを作るよりもそのメッセージを拒否してしまいたい。

内部的な抽象化

内部的な抽象化の中には、今ならそんなことはしないだろうというものもある。また、当時はそうしなかったが、今なら抽象化するだろうというものもある。たとえば、null 終端文字列を長さ/値のペアのかわりに使うことはしないだろう。そうしたほうが標準 C ライブラリで使いやすくなることはわかっていても、である。セキュリティに関する問題ひとつとっても、null 終端文字列を使わない意味がある。逆に言うと、例外処理を C で書こうとは思わない。しかし、一貫性のあるステータスコード体系は作って、それを使うようにするだろう。ルーチンの返り値が `null` や `false` あるいは負の数だったらエラーを意味するなどということはしない。

メールボックスの名前を Unix のユーザー id から切り離すという抽象化は、今ならきっと行うだろう。sendmail を書いていた当時は、Unix のユーザーにメールを送ることしか想定していなかった。今やそんな前提は成立しない。仮に Unix のユーザー管理と同じモデルのシステムであっても、決してメールを受け取ることのないシステムアカウントが存在する。

同じようにしたいところ

もちろん、**うまくいってた**ことだってあったんだよ…

Syslog

sendmail の派生プロジェクトの中でも成功したうちのひとつが、syslog である。sendmail が書かれた当時は、プログラムからログを出力しようとすれば、何かファイルを作ってそこに書き込むしかなかった。その手のログファイルがファイルシステム上に散乱していたのだ。当時は syslog に書き込むのはなかなか難しかった (まだ UDP は存在しなかったので、mpx ファイルとかいうものを使っていた) が、よくやってくれた。しかし、一カ所だけ変更したいところがある。ログに記録されるメッセージの構文にもっと注意を払い、機械可読性を向上させたい。当時の私には、ログ監視ツールの登場を予見できなかったのだ。

書き換えルール

書き換えルールにはいろいろ問題もあるが、今でもきっと採用するだろう (今使われているほど多くはならないだろうが)。タブ文字を使ったことは大きな間違いだった。しかし、ASCII の制約やメールアドレスの構文を考慮すると、何らかのエスケープ文字は必要となる[7]。一般に、パターン置換のパラダイムはうまく機能するし、非常に柔軟である。

ツールに頼りすぎない

先ほどは「もっと既存のツールを活用する」と書いたが、いまどきのランタイムライブラリの多くは、あまり使いたいとは思わない。私見だが、多くのライブラリは肥大化しすぎて危険になっているように感じる。ライブラリの選択は慎重に行うべきだ。再利用することにメリットと、必要以上に高機能なツールを使うことによる問題とのトレードオフになる。これだけは避けようと思っているツールのひとつが XML で、少なくとも設定ファイルを XML にしようとは思わない。XML ファイルの構文は、設定の記述に使うにはごてごてしすぎている。もちろん XML が役立つ場面もあるのだろうが、現状は必要以上に使われすぎている。

コードは **C** で書く

もっと自然に書ける言語、たとえば Java とか C++を使えばいいのではないかと助言してくれる人もいる。C 言語にはいろいろ問題もあるが、それでも私は実装言語に C を使うだろう。まあ個人的な理由もある。C のほうが、Java や C++よりもよく知っているからだ。しかしそれだけではない。いまどきのオブジェクト指向言語にはがっかりさせられているのだ。その多くはメモリ管理に無頓着すぎて、無尽蔵にメモリを使いすぎている。メモリを確保するときにはパフォーマンス上の問題もいろいろ考慮しなければならないのだが、それはここでは書ききれない。sendmail は、内部的にはオブジェクト指向の概念を採用しているところもある (たとえば、マップクラスなど) が、個人的な意見としては、すべてオブジェクト指向化してしまうのは無駄が多く、制約をかけすぎだと思っている。

[7]設定ファイルで Unicode を使うことはあまり広まらないだろうと思っている。

17.9 まとめ

sendmail MTA が誕生したころの世界は、限りなく混乱していた。まるで “西部開拓時代” のようなものだ。電子メールはアドホックな仕組みとしてしか存在しなかったし、今のような標準規格もまだ正式には定まっていなかった。この 31 年の間に “電子メールの問題” の種類も変わった。昔は単に、巨大なメッセージを高負荷な環境でいかにきちんと配送するかというのが問題だったのだが、徐々に spam 対策やウィルス対策のほうが大切になってきた。今や、電子メールの活躍の場はさらに広がっている。電子メールベースのアプリケーションのプラットフォームとして使われることおある。sendmail はこの世界の主力製品に成長した。リスク管理にうるさい企業でさえも、今や電子メールを受け入れている。電子メールは、単なるテキストベースでの一対一のコミュニケーションツールではなく、ミッションクリティカルな部分を担うマルチメディアベースのツールに成長したのだ。

なぜここまで成功できたのか、よくわからない面もある。激動の世界で生き残り続け、かつ成長していくプログラムを、ごく少数のパートタイムの開発者で作り上げるということ。これは、きちんとまとまったソフトウェア開発の方法論があってこそ実現できたことだ。sendmail の成功の要因について、少しでもみなさんに伝わっただろうか。

第 18 章

SnowFlock

Roy Bryant and Andrés Lagar-Cavilla

クラウドコンピューティングは、お手頃な価格で魅力的な計算プラットフォームを提供してくれる。物理的なサーバーを購入して設定するのには時間や手間もかかるし、初期費用も必要だ。クラウドコンピューティングなら、クラウドにある「サーバー」を簡単にレンタルできる。マウスを何度かクリックするだけだし、一時間あたりのコストは 10 セントに満たない。クラウド事業者がこんなに安価でサーバーを提供できている理由は、物理的なコンピューターではなく仮想マシン (VM) を使っているからだ。それを実現する肝となるのが仮想化ソフトウェアで、これは仮想マシンモニター (VMM) と呼ばれている。物理的なマシンをエミュレートする仕組みだ。各ユーザーはそれぞれ安全に隔離された「ゲスト」VM を利用する。ふつうは複数のゲストが一台の物理マシン (「ホスト」) を共有しているのだが、ありがたいことにユーザーはそれを気にせずに済む。

18.1　SnowFlock のご紹介

クラウドは、アジャイルな組織にとってはとてもありがたいものである。物理サーバーの場合を考えてみよう。まずサーバー購入の稟議書を書いて承認を得て (これ、時間がかかるよね)、それから発注。そして、サーバーが発送されてきたら OS やアプリケーションをインストールして設定をする。その間、実際のユーザーはずっと待ち続けることになる。クラウドを使えば、実際に使えるようになるまで何週間も待たされることもない。新しいスタンドアロンのサーバーを、ほんの数分あれば自分の手で作れるのだ。

残念ながら、それ単体でやっていけるクラウドサーバーはほとんどない。お手軽インストールの従量課金モデルということもあって、クラウドサーバーは一般的に、同じように設定されたサーバー群の一員となることが多い。これらを使って、並列コンピューティングやデータマイグレーションそしてウェブページの提供などの動的かつスケーラブルな作業をこなすことになる。同じ静的テンプレートから新たなインスタンスを繰り返し起動することになるので、商用クラウドサービスでは真のオンデマンド利用を保証することができない。いった

んサーバーのインスタンスを作ったら、クラスタのメンバー管理をしたりブローカーに新たなサーバーを追加したりといった作業が必要となる。

SnowFlock は、これらの問題に対応するために VM Cloning を利用する。このときに使うのは、私たちが提案するクラウド API だ。アプリケーションのコードがごく普通にシステムコールを使って OS のサービスを起動するのと同じく、SnowFlock も同様のインターフェイスでクラウドサービスを起動できるようになったのだ。SnowFlock の VM Cloning やリソース割り当て、クラスタ管理、アプリケーションのロジックなどをプログラムで取りまとめられるようになり、単一の論理操作として扱えるようになった。

VM Cloning をコールすると、クローニングの時点での親 VM とまったく同じ内容で複数のクラウドサーバーのインスタンスを作成する。論理的に、これらのクローンは親のすべての状態を引き継ぐ。OS レベルだけでなくアプリケーションレベルのキャッシュも含めてである。さらに、クローンが自動的に内部のプライベートネットワークに追加されるので、動的でスケーラブルなクラスタにうまく参加できるようになる。個別の VM にまとめた新たな計算機リソースをその場で作れ、必要に応じて動的に活用できる。

現実的に使えるレベルにするには、ただ VM のクローンができるだけではなく、十分に効率的かつ高速でなければばならない。本章では SnowFlock における VM Cloning の実装を取り上げる。さまざまなプログラミングモデルやフレームワークをひとつにまとめ、アプリケーションのランタイムやプロバイダーのオーバーヘッドを最小限に抑え、何十もの新たな VM を 5 秒とかからずに作れるようにした、その方法を説明する。

VM Cloning をプログラムで制御するための API を提供しており、C や C++、Python そして Java のバインディングも用意されている。そのおかげで、SnowFlock は極めて柔軟で幅広く使えるものになっている。私たちはこれまでに、SnowFlock をつかってさまざまなシステムのプロトタイプ実装を成功させてきた。並列処理を要する場面では、多数の物理ホストに対する負荷分散を考慮してワーカー VM のクローン先を明示的に選ぶようにして、すばらしい結果を達成した。Message Passing Interface (MPI) を扱う並列アプリケーションで、専用サーバーのクラスタ環境で実行されることが多いというものがあった。このときは、MPI スタートアップマネージャーに手を加えることで、アプリケーション側を一切変更せずによりよいパフォーマンスとより低いオーバーヘッドを実現した。クローンによる未使用のクラスタを、実行時にオンデマンドで配布できるようにしたのだ。最後にもうひとつ紹介しよう。今までの例とはかなり違う使い方だが、SnowFlock を使って、伸縮自在なサーバー群 (elastic servers) の効率とパフォーマンスを改善したのだ。いまどきのクラウドベースの伸縮自在なサーバー群は、サービスの利用量が急増した場合に必要に応じて新たなワーカーを起動するようになっている。そうではなく、稼働中の VM をクローンすることで、SnowFlock は新たなワーカーの立ち上げを 20 倍も高速化した。また、クローンは稼働中の親サーバーのバッファーも引き継ぐので、まっさらなサーバーを立ち上げるのに比べてより高速にフル稼働状態に持ち込める。

18.2　VMのクローニング

その名が示す通り、VM のクローンは親 VM とほぼ同じものである。ただ、実際には必要最小限の相違点がある。MAC アドレスの衝突みたいな問題を回避するためだ。それについては後で説明する。クローンを作るということは、ローカルディスク全体とメモリの状態をコピーして使えるようにするということだ。ここで、設計上の最初のトレードオフに直面する。状態のコピーは事前に行うべきだろうか。それともその場でオンデマンドで行うべきだろうか。

VM クローニングを実現するためのいちばんシンプルな方法は、標準的な「マイグレーション」機能を使うことだ。マイグレーションの使いどころとして一般的なのは、稼働中の VM を別のホストへの移動だ。今のホストでは負荷に耐えられなくなっただとか、今のホストのメンテナンスで電源断が発生するなどという場合に利用する。VM は純粋にソフトウェアなので、ひとつのデータファイルにまとめることができる。そのファイルを新しいホストにコピーして、少し手を加えた後で起動することになる。これを実現するために、既製の VMM は VM の「チェックポイント」を含めたファイルを作る。ローカルファイルシステムの状態やメモリのイメージ、仮想 CPU(VCPU) のレジスタの状態なども含めたファイルである。マイグレーションのときは、元の VM のかわりに新たなコピーを起動する。ただ、起動プロセスの途中でクローンを作る処理に入り、元の VM は稼働したままにしておく。この「貪欲な」手順では、VM 全体の状態を事前に転送する。そうすることで、初期パフォーマンスを最大化している。というのも、VM 全体の状態が実行開始時にそろっているからだ。この方式のレプリケーションの弱点は、VM 全体のコピーという手間のかかる作業をしてからでないと実行開始できないということ。つまり、インスタンスを作るのにとても時間がかかるということだ。

SnowFlock が採用したのは、それとは正反対の「怠惰な」手順による状態のレプリケーションだ。VM が必要とするであろうすべての内容をコピーするのではなく、SnowFlock は起動するために最低限必要なものだけをとりあえず転送する。その他の部分については、クローン側で必要となったときになってはじめて転送する。この方式には二つの利点がある。まず、起動にかかる時間が最短になるということ。事前に必要な処理が限られているからである。次に、効率がよくなるということ。クローンが実際に使う状態だけをコピーするからである。後者の利点が得られるかどうかは、もちろんクローンの状態に依存する。しかし、メモリの全ページにアクセスしたりローカルファイルシステム上の全ファイルにアクセスしたりするようなアプリケーションなどほとんど存在しない。

とはいえ、怠惰なレプリケーションが万能なわけではなく、それなりの代償を払うことになる。状態の転送を最後の最後まで先送りにするので、実行し続けるためには転送が完了するまで待たなければいけない。この状況は、時分割方式のワークステーションでメモリをディスクにスワップさせるときに似ている。レイテンシーの高いところから状態を取得してくるまでアプリケーションの処理がブロックされてしまうのだ。SnowFlock の場合、このブロックによってクローンのパフォーマンスが多少落ちる。このスローダウンがどの程度の影響を

及ぼすかは、アプリケーションによって異なる。高性能計算アプリケーションに関しては、この問題の影響はほとんどないことが経験上わかっている。しかし、データベースサーバーをクローンした場合などは出だしのパフォーマンスが不十分になるだろう。注意すべきなのは、これはあくまでも一時的な問題だということだ。数分もすれば必要な状態の大半は転送し終えるので、クローンのパフォーマンスは親と同じ程度になる。

余談だが、VM に詳しい人ならこう思うかもしれない。「そんなときって、ライブマイグレーションの最適化機能が使えるんじゃないの？」ライブマイグレーションの最適化とは、元の VM を停止してから新たなコピーを起動するまでのインターバルを短縮することである。これを実現するには、元の VM が動いている間に Virtual Machine Monitor (VMM) が VM の状態をコピーすればよい。これで、元の VM を停止したときには直近で変更があったページだけを転送すればいいことになる。ただ、このテクニックを使ったところで、マイグレーションのリクエストからコピーが立ち上がるまでの所要時間は変わらない。そのため、貪欲な VM クローニングにおけるインスタンス作成時間の短縮には役立たない。

18.3 SnowFlock の手法

SnowFlock は、VM のクローニングを「VM フォーク」というプリミティブで実装している。これは標準的な Unix の `fork` と似ているが、重要な違いがいくつかある。まず、Unix の `fork` は単一のプロセスを複製するが、VM フォークの場合は VM 全体を複製する。メモリやプロセスや仮想デバイス、そしてローカルファイルシステムの状態も含めて複製するのだ。次に、Unix の `fork` は同じ物理ホスト上にコピーをひとつ作るだけだが、VM フォークの場合は複数のコピーを並列で作成できる。最後に、VM フォークは別のサーバー上に作ることもできる。これを利用すれば、クラウド上でのフットプリントを必要に応じて拡大できる。

SnowFlock の鍵となるのが、これらの概念だ。

- 仮想化：VM が計算環境をカプセル化することで、クラウドを作ったりマシンをクローンしたりできるようにする。
- 緩やかな伝搬：VM の状態は、必要になるまでコピーしない。そうすることで、クローンを数秒で立ち上げられるようにする。
- マルチキャスト：クローンした兄弟たちは、VM の状態が同じになっていなければいけない。マルチキャストを使えば、何十ものクローンをすばやく稼働させることができる。
- ページフォルト：まだクローン側に存在しないメモリを使おうとしたときは、処理を止めて親にリクエストをしなければならない。クローンの実行は、必要なページが届くまでブロックする。
- コピーオンライト (CoW)：メモリやディスクスペースのコピーを取ってから上書きする。これで、親 VM が処理を続けつつ、それ以前の状態のコピーをクローンが使えるようになる。

SnowFlock の実装には Xen を利用した。ここで、Xen 特有の用語を念のために紹介しておこう。Xen の環境では VMM のことをハイパーバイザと呼び、一方 VM のことをドメインと呼ぶ。物理マシン (ホスト) ごとに特権ドメイン “domain 0” (dom0) があり、これはホストとその物理デバイスに対する完全なアクセス権を持つ。また、この特権ドメインを使ってゲスト (あるいは “user”)VM を作ることができる。このドメインのことを “domain U” (domU) と呼ぶ。

Xen ハイパーバイザに手を加えて、存在しないリソースへのアクセスがあったときのリカバーをスムーズに行えるようにしたもの。dom0 上で稼働するサポートプロセス群で、存在しない VM の状態を転送するもの。クローン VM 内で実行される OS に対する修正。簡単にまとめると、これらが SnowFlock を構成する。主要なコンポーネントは次の六つだ。

- VM Descriptor：この小さなオブジェクトを使ってクローンの種をまき、VM のベアボーンを保持して必要に応じて実行開始できるようにする。実際の作業をするために必要な馬力はない。
- Multicast Distribution System (`mcdist`)：これは親側のシステムで、VM の状態に関する情報をすべてのクローンに対して効率的に同時配信する。
- Memory Server Process：これは親側のプロセスで、親の状態のコピーを管理する。そして、すべてのクローンに対して必要に応じて `mcdist` 経由で提供する。
- Memtap Process：これはクライアント側のプロセスで、クライアントの代わりに働く。メモリーサーバーと通信をして、まだ存在しないが必要になったページについてのリクエストを出す。
- Clone Enlightenment：クローンの中で動くゲストカーネルは、オンデマンドでの VM 状態の転送を緩和するために VMM にヒントを提供できる。これは必須ではないが、効率を考えるとかなりおすすめだ。
- Control Stack：各物理ホスト上で動くデーモンで、他のコンポーネントをとりまとめる役割を果たす。SnowFlock の親とクローン VM を管理する。

図で説明しよう。図 18.1 は VM のクローン手順を表した図で、四つの主要な手順を示している。(1) 親 VM をサスペンドして構造ディスクリプタを作り、(2) このディスクリプタをすべてのターゲットホストに配布し、(3) ほぼ空っぽの状態のクローンを立ち上げ、(4) オンデマンドで状態を伝搬させる。この図は、`mcdist` によるマルチキャスト分散やゲストの Clone Enlightenment によるヒントの提供も示している。

SnowFlock を試してみようと思ったら、選択肢は次の二種類だ。まずは、ドキュメントつきのオープンソースのコード。これがオリジナルで、トロント大学の SnowFlock 研究プロジェクト [1] で公開されている。ちょっと実運用で試してみたいのでそれに耐えるバージョンが欲しいという場合は、フリーで非商用のライセンスのものが GridCentric Inc. [2] から公開されている。SnowFlock はハイパーバイザに手を入れているし dom0 にアクセスする必要もあるの

[1] `http://sysweb.cs.toronto.edu/projects/1`
[2] `http://www.gridcentriclabs.com/architecture-of-open-source-applications`

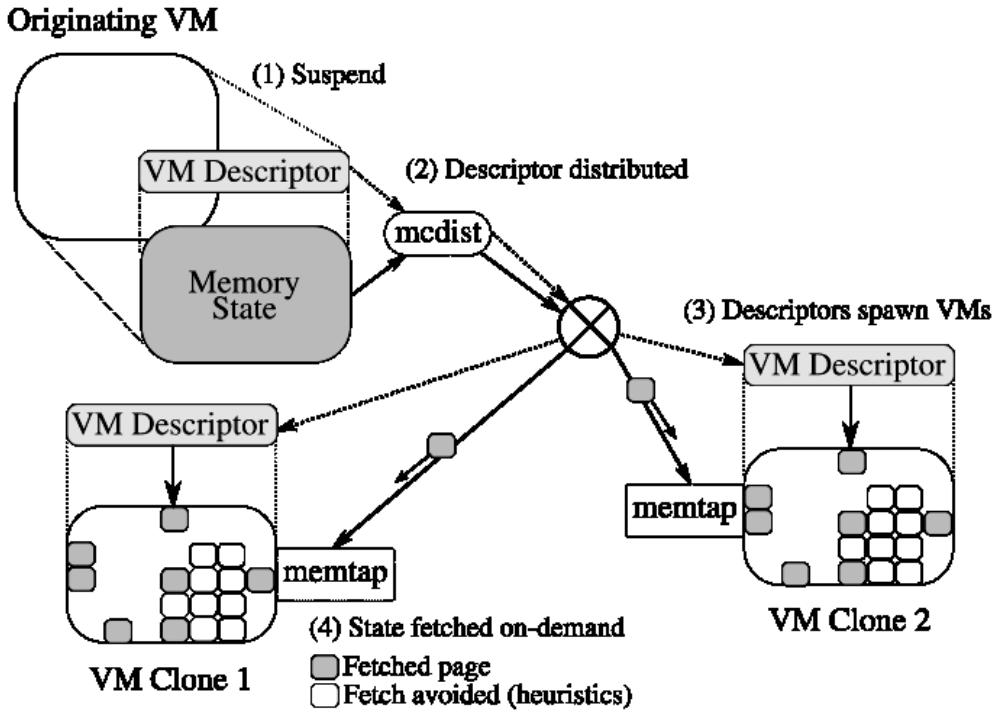



図 18.1: SnowFlock VM レプリケーションのアーキテクチャ

で、SnowFlock をインストールするにはホストマシンの特権アカウントが必要だ。そのため、試すなら自分のハードウェアで使う必要があるだろう。Amazon EC2 などの商用クラウド環境で、ユーザーレベルで試すことはできない。

この後のセクションでは、即座に効率的なクローンを実現するための各パーツについて説明する。ここで説明するすべてのパーツが図 18.2 で示すように組み合わさることになる。

18.4　VM 構造ディスクリプタ

SnowFlock における設計上の重要な決断が、VM の状態のコピーを先送りにして実行時に行うようにしたことだ。言い換えると、VM のメモリのコピーを遅延束縛操作にして、最適化できる余地を増やしたということである。

この決断を実行するための第一歩が、VM の状態を表す構造ディスクリプタの生成だ。これが、クローン VM を作るときに使う種となる。ディスクリプタには、VM を作るために最低限必要な内容だけが含まれており、VM の作成予定を立てて実行できるようになっている。その名の通り、必要最低限な内容とは、基盤となる構造を作るために必要なデータである。SnowFlock の場合の構造とは、Intel x86 プロセッサと Xen だ。したがって、構造ディス

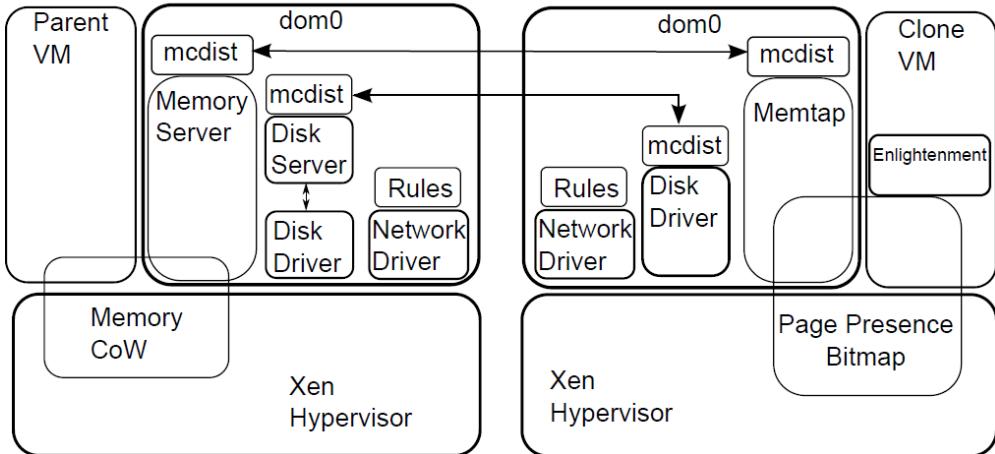



図 18.2: SnowFlock のソフトウェアコンポーネント

クリプタの内容は、ページテーブルや仮想レジスタ、デバイスのメタデータ、時計のタイムスタンプなどといったデータとなる。構造ディスクリプタの内容に関する詳細な情報に関しては [LCWB^{+}11] を参考にした。

構造ディスクリプタには、三つの重要な特徴がある。まず、短時間で作れること。200 ミリ秒で作れるというのも珍しいことではない。次に、小さいこと。一般的には、元の VM のメモリ割り当て量よりも桁が三つくらい少なくなる。つまり、メモリ 1GB の VM なら、1MB 程度になる。最後に、ディスクリプタからのクローン VM の作成が一秒未満で終わること。一般的には 800 ミリ秒程度になる。

もちろん問題もあって、ディスクリプタから VM をクローンした直後の時点では、メモリの状態がほとんど存在しない状態になる。その問題をいかにして解決するのか。それをこの後のセクションで解説する。これはまた、最適化のチャンスをいかに活用するのか、という説明でもある。

18.5　親側のコンポーネント

ある VM のクローンを作ると、元の VM はそのクローン (子ども) たちの親になる。親っていうのは、自分の子どもたちの面倒を見なきゃいけないものだ。そのため、いくつかのサービスを用意して、子どもたちからメモリやディスク状態の配布要求があったときに応えられるようにする。

Memserver プロセス

構造ディスクリプタを作る間は、VM がいったん停止する。これは、VM の状態を安定させるためである。実際に VM を止めて実行スケジュールから外す前に、内部 OS ドライバーが休止状態に入る。クローン側では、新たな VM でこの状態から外界との接続を再開できる。この休止状態を活用するために作ったのが「メモリサーバー」。またの名を `memserver` という。

メモリサーバーは、すべてのクローンに対して必要に応じて親からメモリを与える役割を果たす。その粒度は、x86 のメモリページ (4k バイト) と同じ単位だ。最も単純な形式だと、メモリサーバーはクローンからのリクエストを待ち受けて、一つのクローンに対して一度に 1 ページずつメモリを与える。

しかし、ここで分け与えるメモリは、親 VM の実行に必要なメモリとまったく同じものだ。親側でのメモリの内容も変わり続けるので、もしそれをそのままクローンに与えてしまうと壊れた状態のメモリを渡すことになるかもしれない。クローン側で期待しているのはクローンを作った時点のメモリの状態なのに、実際にやってきた内容はそれとは違う。クローンはきっと戸惑うだろう。カーネルハックの世界だと、これはきっと「スタックトレースを出すための確実な方法」ってことになる。

この問題を回避するための助けとなったのが、昔ながらの OS にある概念であるコピーオンライト (CoW) メモリだ。Xen ハイパーバイザの助けを得れば、親 VM のメモリ内のすべてのページに対する書き込み権限を削除できる。こうすれば、親がどこかのページを書き換えようとしたときにハードウェアページフォルトが発生するようになる。Xen はなぜそうなったのかを知っているので、そのページのコピーを作る。コピーが終われば、親 VM は本来のページへの書き込みを認められて処理を続行できる。一方メモリサーバーには、今後はコピーの方を使うように伝えてコピーのほうを読み込み専用にする。この方式を使えば、クローンした時点でのメモリの状態が残ったままになるのでクローン側で混乱することがなくなる。その一方、親は親で処理を続けられるのだ。CoW によるオーバーヘッドはわずかなものだ。実際、たとえば Linux で新たなプロセスを作るときにも同じ手順が使われている。

Mcdist によるマルチキャスト

クローンは一般的に、いわゆる「運命決定論」という症状に苦しめられている。私たちは、クローンは何かひとつの目的のために作られるものだと考えている。たとえば、X 本の DNA 鎖をデータベースのセグメント Y に割り当てるといったものだ。さらに、複数のクローンをつくったときには、それぞれのクローンが同じことをするものと考える。たとえば同じ鎖 X をデータベースのさまざまなセグメントに割り当てたり、それぞれ別々の鎖を同じセグメント Y に割り当てたりといったものだ。したがって、クローンたちのメモリアクセスは時間局所性が大きくなることは明白だ。同じようなコードを使い、大部分は同じようなデータを扱うことになる。

そこで、その時間局所性を活用するために mcdist を使う。これは、SnowFlock にあわせて自作したマルチキャスト配信システムである。mcdist は、IP マルチキャストを使って同じパケットを複数の受信者に同時配信する。ネットワークハードウェアの並列性を活用して、メモリサーバーへの負荷を軽減する。あるページへの最初のリクエストに対しての応答を全クローンに向けて行うことで、各クローンからのリクエストがその兄弟たちへのプリフェッチとして機能するようになる。兄弟たちのメモリアクセスパターンは似たようなものだからである。

他のマルチキャストシステムとは違い、mcdist には信頼性は要求されないし順序どおりにパケットを配送する必要もない。また、すべての受信者に対してアトミックに応答を返す必要もない。ここでのマルチキャストは厳密に言えば最適化であり、確実に配送しないといけないのは明示的にページを要求したクローンに対してだけだ。そのため、すっきりとシンプルな設計にした。サーバー側は単に応答をマルチキャストするだけ。クライアント側は、もしリクエストに対する応答がなければタイムアウトして、リクエストを再送する。

SnowFlock のための最適化として、次の三つが mcdist に仕込まれている。

- 連続リクエストの検出：時間局所性が発生すると、複数のクローンがほぼ同時に同じページをリクエストすることになる。mcdist サーバーは、そんな場合に最初のリクエストだけを受け付けてその他を無視する。
- フロー制御：受信者は、リクエストと同時に受信レートを送る。サーバー側では、送信レートを調整してクライアントの受信レートに対する重み付けをする。そうしないと、サーバーから大量のページが送られてきてクライアントが処理しきれなくなるだろう。
- 試合終了：サーバーが大半のページを配信し終えたら、ユニキャスト方式に移行する。この時点ではリクエストの大半が再送要求になるので、全クローンに対してページを送りつけなくてもいいからである。

仮想ディスク

SnowFlock のクローンは、(生存期間が短めであることや、生存決定論のせいもあって) めったにディスクを使うことがない。SnowFlock VM の仮想ディスクはルートパーティションを保持し、バイナリやライブラリそして設定ファイルを扱う。重たいデータ処理などは、HDFS (第 9 章) や PVFS などのそれに適したファイルシステムを使って行う。したがって、SnowFlock のクローンがルートディスクから読み込もうとしたときには、カーネルのファイルシステムのページキャッシュを満たすリクエストをすることになる。

それでもなお、クローンから仮想ディスクへのアクセスはできるようにしておく必要がある。ごくまれにでも、そんなアクセスが必要となる場合があるからである。私たちが選んだのは一番抵抗の少ない方法で、ディスクの実装をメモリのレプリケーション設定にできるだけ近づけるようにした。まず、クローンするときにはディスクの状態をいったん固める。つまり、親 VM でディスクを使うときには CoW 方式で行う。ディスクへの書き込みは別の場

所に保存されるということだ。というのも、クローン側からはクローンした時点でのディスクの状態が見えてほしいからである。次に、ディスクの状態は mcdist を使って全クローンにマルチキャストする。メモリと同様に 4 KB 単位の粒度で、時間局所性に関しても同じように考える。最後に、クローン VM に複製されたディスクの状態は永続させない。ディスクの状態はまばらなフラットファイルに格納され、クローンが破棄されるとともに削除される。

18.6　クローン側のコンポーネント

クローンは、構造ディスクリプタから作成した時点では中身がない空っぽの状態だ。というわけで、成長するには親の助けが必要だ。家を飛び出していった子どもたちは、必要なものが手元にないことに気付いた時点で実家に連絡して、「すぐに送って！」と親にお願いする。

Memtap プロセス

クローンができあがるたびにそれにアタッチされるのが memtap プロセスで、これがクローンの命綱となる。クローンのメモリ全体をマップして、必要に応じてその中身を埋める役割を果たす。このプロセスは、Xen ハイパーバイザからの助けを得て動く。クローンのメモリページへのアクセス権を無効にして、あるページへの最初のアクセスがハードウェアフォルトを起こした時点でハイパーバイザが memtap プロセスに処理を回す。

単純に言ってしまえば、memtap プロセスの役割はフォルトが発生したページの配送をメモリサーバーに依頼するだけのことだ。しかしそれ以外にも、もっと複雑なシナリオもある。まず、memtap ヘルパーは mcdist を利用する。つまり、他のクローンが要求したページが任意の時点で到着することがある。そう、非同期プリフェッチってやつだ。次に、SnowFlock VM をマルチプロセッサ VM にすることができる。そうしなければ、いろんな楽しいこともできなかっただろう。これはつまり、複数のフォルトを並列に扱う必要があるということだ。場合によっては同じページが含まれるかもしれない。最後に、最近のバージョンの memtap ヘルパーには、ページ群を明示的にプリフェッチする機能がある。ただし、ページを取得する順序がどうなるかはわからない。mcdist サーバーにそれを保証する機能がないからだ。これらの要素はすべて同時並行性に関する悪夢を引き起こす元であり、実際に私たちもそれを経験した。

memtap の全体的な設計の中心となるのが、ページの存在を表すビットマップだ。このビットマップを作って初期化するのは、構造ディスクリプタがクローン VM を作るときである。このビットマップは一次元のビット配列で、VM のメモリが保持できるページ数と同じだけのサイズになる。Intel プロセッサには、便利なアトミックビット操作インストラクションが用意されている。ビットを立てたり、状態を調べてから設定したりといった操作が、同じマシン内の他のプロセッサに対してアトミックに行えることが保証されるのだ。そのおかげで、大半の場合にロックを回避でき、別の保護ドメインにある別のエンティティ(Xen ハイパーバ

イザや memtap プロセス、そしてクローンしたゲストのカーネル自身など) からのビットマップへのアクセスもできるようになる。

あるページへの最初のアクセスで発生したハードウェアページフォルトを Xen が処理するときに、このビットマップを使って memtap に通知するかどうかを判断する。また、同じくこのビットマップを使って、複数の仮想プロセッサが同じページでフォルトを起こしている場合にキューに入れたりする。Memtap は、到着したページをバッファに格納する。バッファがいっぱいになったり明示的にリクエストしたページが届いたりしたときは VM を一時停止させ、ビットマップを使って不要なページを破棄する。不要なページとは、重複して届いたけれど既に存在するようなページのことだ。残りのページは必要なものなので VM のメモリ内にコピーし、それに合わせてビットマップ上の適切なビットを立てる。

無駄なフェッチを回避する賢いクローン

先ほど言ったとおり、ページビットマップはクローン内で稼働中のカーネルからも見えており、ビットマップを変更するときにロックは不要だ。これは、クローンにとって非常に強力な "enlightenment" ツールとなる。ビットマップをいじって特定のページが存在するように見せかければ、そのページをフェッチさせないようにできるのだ。これはパフォーマンス的に極めて有用で、ページを使う前に完全に上書きしてしまうようなときにも安全に行える。

このような状況になってフェッチを回避するようなことは非常によくある話だ。カーネル内でのすべてのメモリ確保 (vmalloc や kzalloc、get_free_page、ユーザー空間での brk など) は、結局のところカーネルのページアロケータが処理する。一般的に、ページをリクエストするのは中間アロケータで、ここではよりきめ細やかな塊を管理する。slab アロケータ、そしてユーザー空間プロセス用の glibc malloc アロケータなどである。しかし、明示的なアロケーションであっても暗黙のアロケーションであっても、これだけは常に正しい。そのページに何が含まれるかなんて、誰も気にしない。ページの内容なんてどうにでも上書きできるからだ。じゃあなぜそんなページをフェッチするんだって？理由なんかない。経験上、そんなページのフェッチは回避するのが大いに有用なのだ。

18.7 VM クローニングアプリケーションのインターフェイス

ここまでは、VM を効率的にクローンするための内部構造に焦点を合わせてきた。自分たちの世界にだけ浸っているわけにもいかないので、ここらでシステムを使う側にも目を向けよう。アプリケーション側の視点で話を進める。

API の実装

VM クローニングをアプリケーションから行うには、シンプルな SnowFlock API を使う。図 18.1 に示すようなものだ。クローニングは基本的には二段階のプロセスである。まず最初

sf_request_ticket(n)	n 個のクローン用に割り当てを要求する。m≤n 個のクローン用の割り当てを示す ticket を返す。
sf_clone(ticket)	ticket の割り当てにしたがってクローンを作る。クローンの ID(0≤ID<m) を返す。
sf_checkpoint_parent()	親 VM の、イミュータブルなチェックポイント C を作る。これを使えば、その後いつでもクローンを作れるようになる。
sf_create_clones(C, ticket)	sf_clone と同じだが、チェックポイント C を利用する。対応する sf_checkpoint_parent() を実行した時点の状態からクローンの実行が始まる。
sf_exit()	子 VM(1≤ID<m) を終了する。
sf_join(ticket)	ticket 内のすべての子が sf_exit を呼ぶまで、親 (ID = 0) をブロックする。この時点ですべての子が終了し、ticket が破棄される。
sf_kill(ticket)	親限定で、ticket を破棄して関連するすべての子を終了させる。

表 18.1: SnowFlock VM クローニング API

に、クローンインスタンス用の割り当てを要求する。しかし、システムのポリシーが原因で、要求したよりも少なめの割り当てしかもらえないこともあり得る。次に、割り当て分を使って VM をクローンする。ここでポイントとなるのが、VM は単一の操作に注力するものだという前提である。VM のクローニングが適しているのは、単一用途の VM の場合である。ウェブサーバーやレンダリングファームなどがその一例だ。100 プロセスのデスクトップ環境で複数のアプリケーションが同時に VM のクローニングをしたりなどすれば、大混乱になってしまう。

この API は、単にメッセージをマーシャルして XenStore にポストするだけである。XenStore とは共有メモリの低スループットインターフェイスで、Xen がプレーンなトランザクションを制御するために使っている。SnowFlock Local Daemon (SFLD) がハイパーバイザ上で稼働し、このリクエストを待ち受ける。受けとったメッセージをアンマーシャルして実行し、リクエストを戻す。

プログラムから VM クローニングを直接制御するときには API を利用する。用意されているのは C や C++、Python、そして Java 用の API だ。シェルスクリプトからプログラムを実行できない場合は、代わりに使えるコマンドラインスクリプトが提供されている。MPI のような並列プログラミングフレームワークに API を組み込むこともできる。そうすれば、MPI のプログラムから SnowFlock を使えるようになり、ソースへの変更も不要となる。ウェブサー

バーあるいはアプリケーションサーバーの前に置いたロードバランサーも、この API を使えば自身の管理するサーバーをクローンできる。

SFLD は、VM クローニングリクエストの実行の調整役となる。SFLD が構造ディスクリプタを作ってそれを送信し、クローン VM を作成し、ディスクとメモリサーバーを立ち上げ、ヘルパープロセス `memtap` を立ち上げる。これらはちょっとした分散システムであり、物理クラスタ内の VM を管理する役割を果たす。

SFLD は、割り当ての判断を中央の SnowFlock Master Daemon (SFMD) に任せる。SFMD は単に、適切なクラスタ管理ソフトウェアとのインターフェイスになるだけのものである。こんなところで車輪の再発明をする必要もないので、リソース割り当てや領域の分配そしてポリシーなどは Sun Grid Engine や Platform EGO といった適切なソフトウェアに任せることにしたのだ。

欠かせない変異

クローニングで作ったクローン VM のプロセスの大半は、自分がもはや親ではなく親のコピーであるということを知るすべがない。たいていの場合はそれでもうまく動くし問題はない。結局のところ、OS の最大の目的はアプリケーションをローレベルのあれこれ (ネットワーク上での ID など) から切り離すことである。しかし、移行をスムーズに進めるにはいくつかの仕組みが必要となる。問題の本質は、ネットワーク上でのクローンの一意性を管理することだ。衝突や混乱を回避するために、クローン処理の中でちょっとした変異が必要となる。また、これらの手入れで上位レベルでの調整を余儀なくされることもあるので、フックを導入して必要なタスクをユーザーが設定できるようにする。たとえば、クローンのネットワーク情報に基づいてマウントしているネットワークファイルシステムの再マウントなどといったタスクである。

クローンは、もともとその存在が予期されていなかった世界に生まれてきたものだ。親 VM は、もともと管理されたネットワークの一員であった。たいていは DHCP サーバーの配下にあっただろう。あるいはそれ以外の何かかもしれない。システム管理者が飯を食う種などいくらでもある。必ずしも柔軟性のないシナリオを想定するのではなく、私たちは親とすべてのクローンたちを同じプライベート仮想ネットワーク内に置くことにした。同じ親からのクローンたちにはすべて一意な ID を割り当て、プライベートネットワーク内での IP アドレスは、クローンの際に ID をもとに自動的に設定する。これでシステム管理者の介入は一切不要となり、IP アドレスの衝突も発生しなくなる。

IP アドレスの再設定は、仮想ネットワークドライバ上に仕込んだフックが直接行う。しかし、それとは別に、DHCP のレスポンスを自動生成するドライバも用意している。したがって、どんなディストリビューションを選んだとしても、仮想ネットワークインターフェイスはゲスト OS に対して適切な IP アドレスを割り振ることが保証される。スクラッチから再起動しても同様である。

別々の親からのクローンがお互いに相手の仮想プライベートネットワークと衝突してしまうことを防ぐため—そしてお互いの DDoS 攻撃を防ぐため—に、クローンの仮想ネットワークはイーサネットレベル (レイヤー 2) で隔離されている。Ethernet MAC OUI [3] を乗っ取り、そこをクローン専用にしているのだ。この OUI は親 VM の役割となる。VM の ID をもとにしてその IP アドレスを決めるのと同様に、OUI 部分以外の MAC アドレスも ID にもとづいて決める。仮想ネットワークドライバは、VM が自分自身の ID にもとづいて与えられたと信じている MAC アドレスを翻訳し、仮想プライベートネットワークと外部との間のトラフィックをフィルタして別の OUI にする。これは `ebtables` が使っているのと同じ方式だが、実装はそれよりもはるかにシンプルだ。

クローンはクローン同士でしか通信できないようにするというのもおもしろいが、それでは不十分だ。時にはクローンからインターネットに HTTP レスポンスを返したいこともあるだろうし、公開されているデータリポジトリをマウントしたいこともあるだろう。親やクローンの中からいくつかを選んで、専用のルーター VM にすることもできる。この小さな VM の役割は、クローンたちとインターネットとの間のトラフィックのファイアウォールであったり流量制限であったり NAT であったりする。また、外部からの通信は親 VM の wel-known ポートだけに限定している。ルーター VM は軽量だが、ネットワークトラフィックを一元管理する単一点となる。そのため、スケーラビリティに大きな制限ができてしまう。同じようなネットワークルールを分散形式で適用し、クローン VM が稼働する各ホスト上に置くこともできるだろう。そんな実験的なパッチは、まだリリースしていない。

SFLD は ID を割り当て、どんな設定にすべきかを仮想ネットワークドライバに伝える。内部的な MAC アドレスおよび IP アドレス、DHCP のディレクティブ、ルーター VM の設定、フィルタリング規則などである。

18.8　結論

Xen ハイパーバイザに手を入れて VM の状態の転送を遅らせるようにしたことで、SnowFlock は稼働中の VM のクローンを何十個も作れるようになった。それもたった数秒でだ。つまり、SnowFlock における VM のクローニングとは、瞬時にその場で行えるものだということになる。これはクラウドの使い勝手を大きく向上させた。クラスタ管理は自動化できるし、アプリケーションのプログラムからクラウドのリソースを操作することもできるようになった。SnowFlock はまた、クラウドのアジリティも向上させた。VM の立ち上げ速度は 20 倍高速化したし、新しく作った VM のパフォーマンスも向上した。親の環境のまさに稼働中の状態 (メモリやアプリケーションのキャッシュなど) をそのまま引き継げたからである。SnowFlock がパフォーマンスを向上させた鍵は、ヒューリスティックを活用して無駄なページフェッチを回避したことと、同じ親を持つクローン同士がマルチキャストシステムで状態をプリフェッチできるようにしたことだ。これらすべての要因となったのが、いくつかの実証済みのテク

[3]Organizational Unique ID とは、ベンダーに割り当てられた MAC アドレス範囲のことだ。

ニックをうまく適用したことやそこに加えたちょっとした小技、そして業務レベルでのデバッグだ。

SnowFlock を通じて得た大切な教訓が二つある。第一の教訓は、過小評価されることが多い KISS 原則である。クローンの起動時に発生する大量のページ要求に対応するため、当初は込み入ったプリフェッチ技術を実装しようとしていた。以外にも、そんなことはしなくてもよかったのだ。たったひとつの原則「必要になった時点でメモリを持ってくる」に従うだけで、システムのパフォーマンスは高負荷時でも非常に良好だった。シンプルに保つメリットを示すもうひとつの例が、ページの存在を表すビットマップだ。データ構造をシンプルにしておき、きっちりとアトミックなアクセス方式を使うことで、同時並行性に関するおぞましい問題をかなりシンプルにできた。また、複数の仮想 CPU が同じページの更新を要求する問題も、マルチキャストを使ってページを非同期配信することで解決できた。

第二の教訓は、規模は嘘をつかないということ。言い換えれば、システムの規模が倍になるたびに新たなボトルネックが発覚してシステムが大変なことになってしまうのでその準備が必要だということだ。これは、先ほどの教訓と密接に関連する。シンプルでエレガントなソリューションは規模が拡大してもうまく機能するし、負荷が高まってもいやなことにならずに済ませられる。この法則の主要な例が、`mcdist` システムだ。大規模環境でのテストにおいて、TCP/IP ベースのページ配信機構はクローン数百台程度でぶざまに失敗してしまう。mcdist は、窮屈なうまく定義したロールのおかげで成功をおさめた。クライアントは自分のページだけを気にすればよくて、サーバーは全体的なフロー制御だけを管理すればよいというものだ。`mcdist` を控え目でシンプルなものに保つことで、SnowFlock はとてもうまい具合にスケールできるようになった。

より詳しく知りたくなった人は、トロント大学のサイト [4] に行ってみよう。学術論文や、GPLv2 でライセンスされたオープンソースのコードがある。また GridCentric[5] には、業務レベルでの使用に耐える実装がある。

[4]http://sysweb.cs.toronto.edu/projects/1
[5]http://www.gridcentriclabs.com/

第19章

SocialCalc

Audrey Tang

スプレッドシートの歴史は、かれこれ30年以上にもなる。Dan Bricklinが最初のスプレッドシートプログラムであるVisiCalcの構想をたてたのが1978年、そして実際に出荷されたのが1979年だ。当初のコンセプトは極めて単純なものだった。縦横両方向に好きなだけ拡張できる表があって、そのセルにテキストや数値や数式を入力する。ただそれだけ。数式は、ごく普通の算術演算子やさまざまな組み込み関数で構成されており、数式の中では他のセルの現在の値を利用できる。

考え方自体はシンプルだけれど、適用範囲は幅広い。会計や在庫管理、あるいは一覧表の管理などはほんの一例に過ぎない。そこには無限の可能性があった。というわけで、VisiCalcはパソコン時代の最初の「キラーアプリ」になったのだ。

その後数十年の間にLotus 1-2-3やExcelなどの後継者が登場した。機能はどんどん改良されていったが、もととなるメタファーは同じままだ。大半のスプレッドシートはディスク上のファイルとして保存され、ファイルを開いたときにメモリに読み込まれて、それを編集する。こういったファイルベースのモデルでは、複数人でのコラボレーションは極めて難しい。

- 各ユーザーが、スプレッドシートの編集ソフトをインストールする必要がある。
- メールを使うにせよ共有フォルダを使うにせよバージョン管理システムを使うにせよ、余計なオーバーヘッドが増える。
- 変更履歴の追跡がしにくい。たとえば、Excelには書式の変更やセルのコメントの履歴を記録する機能がない。
- テンプレートの書式や数式を変更すると、そのテンプレートを使っているスプレッドシートファイルにも面倒くさい変更作業が必要になる。

ありがたいことに、新たなコラボレーションモデルの登場によって、この問題をエレガントかつシンプルに解決できるようになった。それがWikiである。Ward Cunninghamが1994年に発案したWikiは、2000年代はじめにWikipediaによって広まった。

Wikiモデルでは、ファイルのかわりにサーバーでページをホストする。このページはブラウザ上で編集でき、特別なソフトウェアを入れる必要はない。ハイパーテキストのページな

ので他のページへのリンクも簡単で、他のページを埋め込んで大きなページを作ることさえできる。デフォルトでは誰もがページの表示と編集が可能で、編集履歴は自動的にサーバーが管理する。

Wiki モデルに触発されて Dan Bricklin は WikiCalc の開発を始めた。2005 年のことである。その狙いは、管理しやすくて多人数で編集できるという Wiki の特徴と、ビジュアルな書式設定や計算機能というスプレッドシートのメタファーを組み合わせることだった。

19.1 WikiCalc

WikiCalc (図 19.1) の最初のバージョンにはこんな機能があり、これらは、当時のスプレッドシートソフトウェアとは一線を画すものだった。

- プレーンテキストや HTML、そして Wiki 記法によるテキストデータのマークアップ。
- Wiki 記法には、リンクや画像を挿入したり他のセルの値を参照したりするコマンドも含まれている。
- セルの数式内で、他のウェブサイトで公開されている別の WikiCalc ページの値を参照できる。
- 出力を他のウェブページに埋め込むことが可能。静的にも埋め込めるし、生きたデータとして動的に埋め込むこともできる。
- セルの書式設定で、CSS の属性やクラスにアクセスできる。
- すべての編集操作を監査証跡として記録できる。
- Wiki と同様に同一ページの各バージョンを保存でき、ロールバックも可能。

図 19.1: WikiCalc 1.0 のインターフェイス

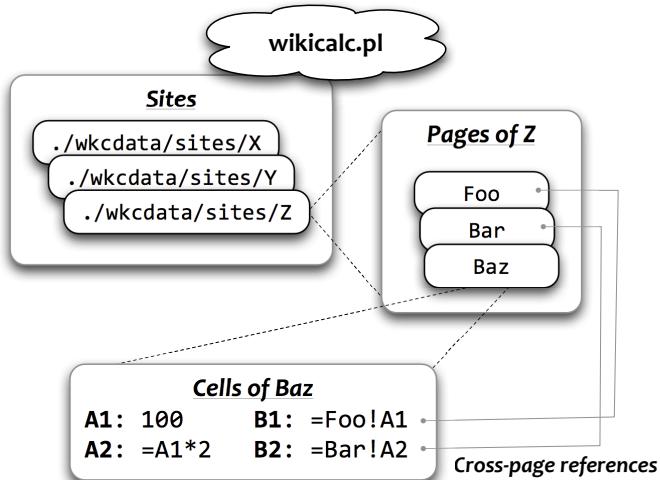



図 19.2: WikiCalc のコンポーネント

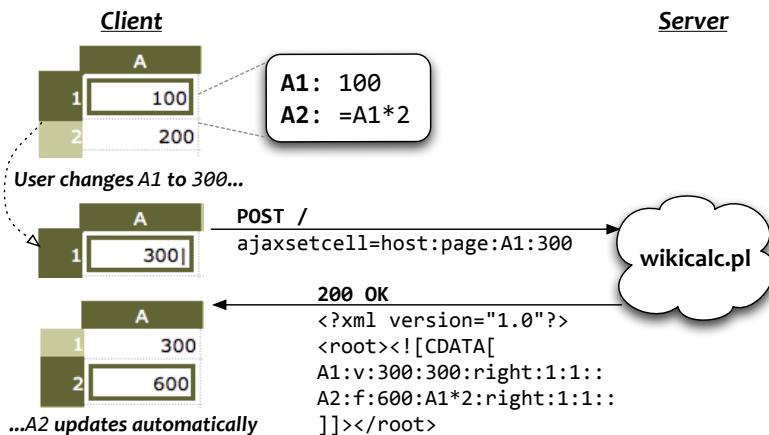



図 19.3: WikiCalc のフロー

WikiCalc 1.0 の内部アーキテクチャ(図 19.2) と情報の流れ (図 19.3) は意図的にシンプルにしているが、それでも強力なものだった。複数の小さなスプレッドシートを組み合わせてマスタースプレッドシートを作る仕組みは特に有用だった。たとえば、各販売員が個別のスプレッドシートに売り上げを記録しているという状況を考えてみよう。地域のマネージャーは各メンバーの売り上げをまとめて地域ごとのスプレッドシートを作ることができるし、販売担当の VP は各地域の売り上げをまとめてトップレベルのスプレッドシートを作れる。

販売員が自分のスプレッドシートで売り上げを更新すると、それをとりまとめた他のスプレッドシートにもすぐにその更新が反映される。とりまとめたスプレッドシート上から詳細

が知りたければ、単にクリックするだけでその元となったスプレッドシートを閲覧できる。この機能があるおかげで、ひとつの数値を複数の場所で管理するなどという冗長かつ間違えやすい状況は回避でき、すべての情報が最新であることが保証できる。

数値機を再計算して最新状態を保つために WikiCalc はシンクライアント方式を採用しており、状態に関するすべての情報をサーバー側で管理している。各スプレッドシートは、ブラウザ上では`<table>`要素として表示される。セルを編集するときには `ajaxsetcell` コールをサーバーに送り、サーバーは、更新すべきセルをブラウザに通知する。

当然ながら、この方式はブラウザとサーバーの間で高速な通信ができることが前提となる。レイテンシーが高くなると、セルの内容を更新したり新たなコンテンツを表示したりするときに “Loading…” メッセージが頻発するようになる。ちょうど図 19.4 のような感じだ。数式を対話的に編集して結果がどう変わるかをすぐ確認したいといったユーザーにとって、これは大きな問題となる。

図 19.4: 読み込み中のメッセージ

さらに、`<table>`要素の行と列の数がスプレッドシートと等しくなるので、100×100 のグリッドを作ると 10,000 の`<td>` DOM オブジェクトを作ることになる。これはブラウザのメモリリソースを大量に消費し、さらにページのサイズにも制約が発生してしまう。

これらの欠点もあって、WikiCalc はローカルホスト上でスタンドアロンで実行するには便利だが、ウェブベースのコンテンツ管理システムに組み込んで使うのは非現実的だった。

2006 年に、Dan Bricklin は Socialtext 社と組んで SocialCalc の開発に取りかかった。これは、WikiCalc を JavaScript でゼロから書き直したもので、元の Perl のコードの一部を元にしている。

この書き直しの狙いは、大規模な分散環境でのコラボレーションを行うことであり、デスクトップアプリケーションに負けないような見た目や操作性を提供することであった。またそれ以外にも、こういった目標を掲げていた。

- 何十万ものセルを扱えること。
- 編集操作の所要時間を高速化すること。
- クライアント側で監査証跡や undo ／ redo スタックを持つこと。
- JavaScript と CSS を活用して、本格的なレイアウト機能を提供すること。

• JavaScript を今まで以上に利用しつつも、クロスブラウザ対応を実現すること。

三年の開発期間と数多くのベータリリースを経て、Socialtext 社は 2009 年に SocialCalc 1.0 をリリースした。これは、当初掲げた設計の目標をクリアしたものだった。さあ、SocialCalc システムのアーキテクチャを見ていこう。

19.2 SocialCalc

図 19.5: SocialCalc のインターフェイス

図 19.5 と図 19.6 は、それぞれ SocialCalc のインターフェイスとクラスを示したものだ。WikiCalc に比べてサーバー側の役割が大幅に減っている。サーバーの唯一の責務は、HTTP GET リクエストに応えてスプレッドシート全体を保存形式にシリアライズしたものを提供することだ。いったんブラウザにデータが渡ってしまった後は、すべての計算や変更の追跡そしてユーザーとの対話処理などが JavaScript で行われるようになった。

JavaScript のコンポーネントは階層型の MVC (Model/View/Controller) 形式で作られており、各クラスがそれぞれ一つの役割に焦点を合わせている。

Sheet はデータモデルで、スプレッドシートのメモリ内での構造を表す。座標から *Cell* オブジェクト (個々のセルを表す) までの辞書を含む。空のセルのエントリは不要なので、空のセルはメモリを消費しない。

Cell はセルの内容や書式を表す。よく使うプロパティの一部を表 19.1 に示す。

RenderContext はビューを実装する。シートを DOM オブジェクトとしてレンダリングする役割を果たす。

TableControl はメインコントローラーで、マウスやキーボードのイベントを受け付ける。スクロールやリサイズといったビューのイベントを受け取ると、それに関連する *RenderContext* オブジェクトを更新する。シートの中身に影響を及ぼす更新イベントを受け取ると、新たなコマンドをシートのコマンドキューに追加する。

SpreadSheetControl はトップレベルの UI を管理する。ツールバーやステータスバー、ダイアログボックスそしてカラーピッカーも含む。

図 19.6: SocialCalc のクラス図

SpreadSheetViewer はもうひとつのトップレベル UI で、読み込み専用のインタラクティブビューを提供する。

私たちが採用したのは、最小限のクラスベースのオブジェクトシステムとシンプルな合成/委譲の仕組みであり、継承やオブジェクトのプロトタイプは使っていない。すべてのシンボルは `SocialCalc.*`名前空間の中にあるので、名前が衝突する心配はない。

シートへの更新はすべて `ScheduleSheetCommands` メソッドを経由する。このメソッドは、編集操作を表すコマンド文字列を受け取る (よく使われるコマンドの一部を表 19.2 に示す)。SocialCalc を組み込んで使うアプリケーションが独自のコマンドを追加することもできる。名前付きコールバックを `SocialCalc.SheetCommandInfo.CmdExtensionCallbacks` オブジェクトに追加して、`startcmdextension` コマンドでそれを実行すればよい。

datatype	t
datavalue	1Q84
color	black
bgcolor	white
font	italic bold 12pt Ubuntu
comment	Ichi-Kyu-Hachi-Yon

表 19.1: セルの内容と書式

```
set     sheet defaultcolor blue       erase   A2
set     A width 100                   cut     A3
set     A1 value n 42                 paste   A4
set     A2 text t Hello               copy    A5
set     A3 formula A1*2               sort    A1:B9 A up B down
set     A4 empty                      name    define Foo A1:A5
set     A5 bgcolor green              name    desc   Foo Used in formulas like SUM(Foo)
merge   A1:B2                         name    delete Foo
unmerge A1                            startcmdextension UserDefined args
```

表 19.2: SocialCalc のコマンド

19.3 コマンド実行ループ

反応速度を向上させるために、SocialCalc はすべての再計算や DOM の更新をバックグラウンドで行う。そのため、ユーザーが複数のセルを編集していたとしても、エンジンがそれをきちんと追いかけて変更した順にそれをコマンドキューに送る。

コマンドを実行している間は、`TableEditor` オブジェクトの busy フラグが true になる。フラグが立っている場合、それに続くコマンドは `deferredCommands` キューに送られる。このキューに入ったコマンドは、順序通りに実行されることが保証される。図 19.7 のイベントループ図が示すとおり、Sheet オブジェクトは `StatusCallback` イベントを送出してコマンドの実行状況をユーザーに通知する。実行状況は、次の四段階に分かれる。

ExecuteCommand: 開始時に `cmdstart` を送り、コマンドの実行が完了したときに `cmdend` を送る。そのコマンドが間接的にセルの値を変更した場合は *Recalc* ステップに入る。そうではなく、そのコマンドが画面上のセルの見た目を変更した場合は *Render* ステップに入る。どちらにもあてはまらない場合 (`copy` コマンドなど) は *PositionCalculations* ステップまで飛ぶ。

図 19.7: SocialCalc のコマンド実行ループ

Recalc (**必要に応じて**): 開始時に `calcstart` を送り、セルの依存関係チェインをチェックしている間は 100ms ごとに `calcorder` を送る。チェックが終われば `calccheckdone` を送り、影響するすべてのセルが再計算後の値を受け取った時点で `calcfinished` を送る。このステップの後には必ず *Render* ステップが続く。

Render (**必要に応じて**): 開始時に `schedrender` を送り、`<table>`要素が新しい書式のセルで更新された時点で `renderdone` を送る。このステップの後には必ず *PositionCalculations* ステップが続く。

PositionCalculations: 開始時に `schedposcalc` を送り、スクロールバーや編集可能セルのカーソルなど `TableEditor` のビジュアルコンポーネントの更新を終えた時点で `doneposcalc` を送る。

そのコマンドが実行されたという情報はすべて保存されるので、ごく自然にあらゆる操作の監査証跡が得られることになる。`Sheet.CreateAuditString` メソッドを使えば改行区切りの文字列として監査証跡が得られる。各コマンドが 1 行で表される形式だ。

`ExecuteSheetCommand` はまた、各コマンドを実行するときに、その取り消し用のコマンドも作る。たとえば A1 セルの中身が "Foo" のときにユーザーが `set A1 text Bar` コマンドを実行したとしよう。このときに作られる取り消し用コマンドは `set A1 text Foo` で、これが undo スタックに積まれる。ユーザーが Undo をクリックすると、このコマンドを実行して A1 を元の状態に戻すというわけだ。

19.4 表エディタ

では次に、TableEditor のレイヤーを見ていこう。ここでは `RenderContext` の画面上での座標を計算したり、水平方向および垂直方向のスクロールバーをふたつの `TableControl` のインスタンスで管理したりする。

図 19.8: TableControl のインスタンスがスクロールバーを管理する

`RenderContext` クラスが扱うビューレイヤーも、WikiCalc とは異なる設計になっている。個々のセルを`<td>`要素にマップするのではなく、まずはブラウザの表示領域全体に広がる固定サイズの`<table>` を作って、そこに事前に`<td>`要素を埋めていったのだ。

カスタム描画したスクロールバーを使ってユーザーがスプレッドシートをスクロールさせると、事前に描画済みの`<td>`の `innerHTML` を動的に更新する。つまり、ほとんどの場合は`<tr>`や`<td>` を作ったり消したりすることが不要になるというわけだ。このおかげで、応答速度は大きく向上した。

`RenderContext` は今見えている領域しか描画しないので、Sheet オブジェクトのサイズをいくら大きくしてもパフォーマンスには影響しない。

TableEditor には CellHandles オブジェクトも含まれている。このオブジェクトは、現在編集可能なセル (ECell) の右下に表示される fill/move/slide といったラジアルメニューを実装するものだ。その様子を図 19.9 に示す。

図 19.9: 現在編集可能なセル (ECell)

入力ボックスを管理するクラスは InputBox と InputEcho のふたつである。前者はグリッド上の編集行を管理し、後者はユーザーがタイプした内容のプレビューを ECell の上にかぶせるレイヤーだ (図 19.10 を参照)。

図 19.10: 入力ボックスは二つのクラスで管理する

通常は、SocialCalc エンジンとサーバーとの通信が必要になる場面は限られている。スプレッドシートを編集用に開くときとそれをサーバーに保存するときだけだ。このために用意されているのが Sheet.ParseSheetSave で、これは保存フォーマット文字列をパースして Sheet オブジェクトにする。一方 Sheet.CreateSheetSave メソッドは、Sheet オブジェクトをシリアライズして保存フォーマットに戻す。

数式では、別の場所にあるスプレッドシートの値を URL を指定して参照することもある。recalc コマンドは参照先の外部スプレッドシートを再取得し、もう一度 Sheet.ParseSheetSave でパースしてキャッシュに保存する。そうすれば、同じリモートスプレッドシートにある別のセルを参照するときにも再取得が不要になる。

19.5 保存フォーマット

保存フォーマットは標準的な MIME multipart/mixed 形式で、四つの text/plain; charset=UTF-8 パートを組み合わせたものだ。それぞれのパートには、コロンで区切られたデータフィールドが改行区切りで複数行含まれている。内容は次のとおりである。

- meta パートには、他のパートの型の一覧がある。
- sheet パートには、各セルの書式や値、各カラムの幅 (デフォルト以外を使っている場合)、シートのデフォルト書式の一覧が含まれ、その後にシート内で使われているフォントや色、罫線の一覧が続く。
- オプションの edit パートには、TableEditor の編集状態を保存する。ECell の最新の位置や行／列ペインの固定サイズなどである。
- オプションの audit パートには、前回の編集セッションで実行したコマンドの履歴が含まれる。

図 19.11 に示したスプレッドシートには三つのセルが含まれる。A1 の値は 1874 でこれが ECell であり、A2 には数式 2^2*43 が書かれている。また A3 には数式 SUM(Foo) が書かれており、ボールドになっている。名前で指定した範囲 Foo が指すのは A1:A2 だ。

図 19.11: 三つのセルからなるスプレッドシート

このスプレッドシートをシリアライズした保存フォーマットは、このようになる。

```
socialcalc:version:1.0
MIME-Version: 1.0
Content-Type: multipart/mixed; boundary=SocialCalcSpreadsheetControlSave
--SocialCalcSpreadsheetControlSave
Content-type: text/plain; charset=UTF-8

# SocialCalc Spreadsheet Control Save
version:1.0
part:sheet
part:edit
part:audit
--SocialCalcSpreadsheetControlSave
Content-type: text/plain; charset=UTF-8

version:1.5
cell:A1:v:1874
cell:A2:vtf:n:172:2^2*43
cell:A3:vtf:n:2046:SUM(Foo):f:1
sheet:c:1:r:3
font:1:normal bold * *
name:FOO::A1\cA2
--SocialCalcSpreadsheetControlSave
Content-type: text/plain; charset=UTF-8
```

```
version:1.0
rowpane:0:1:14
colpane:0:1:16
ecell:A1
--SocialCalcSpreadsheetControlSave
Content-type: text/plain; charset=UTF-8

set A1 value n 1874
set A2 formula 2^2*43
name define Foo A1:A2
set A3 formula SUM(Foo)
--SocialCalcSpreadsheetControlSave--
```

このフォーマットは人間が読めるように作られているが、同時に機械的に生成しやすいようにも作られている。そのおかげで、Drupal の Sheetnode を使えば、この保存フォーマットを Excel 形式 (`.xls`) や OpenDocument 形式 (`.ods`) などの他のフォーマットに変換できるようになった。

さあ、これで SocialCalc の各パーツがどのように機能するのかがわかった。ここからは、SocialCalc を拡張する実例を二つ見ていこう。

19.6 リッチテキストの編集

まず最初に見る例は、SocialCalc のテキストセルを拡張したものだ。Wiki 記法を使って、表エディタの中でリッチテキストの表示を実現している (図 19.12)。

図 19.12: 表エディタにおけるリッチテキストのレンダリング

この機能を SocialCalc に追加したのはバージョン 1.0 のリリース直後のことだった。これは、画像やリンクの挿入そしてテキストのマークアップをすべて同じ方法でやりたいという要望に応えたものである。Socialtext 社は既にオープンソースの Wiki プラットフォームを持っていたので、その構文を SocialCalc で使うのはごく自然な流れだった。

この機能を実装するためには text-wiki の textvalueformat 用のカスタムレンダラーが必要で、それを使うテキストセル側でもデフォルトフォーマットの変更が必要だった。

textvalueformat なんて知らないよだって？まあ続きをよんでくださいな。

型と書式

SocialCalc では、各セルが datatype と valuetype を保持している。テキストや数値が入っているセルはそれぞれ text/numeric の値型に対応し、datatype="f"である数式セルは数値あるいはテキストの値を生成する。

先ほどの Render ステップを思い出してみよう。Sheet オブジェクトは個々のセルから HTML を生成していた。このとき、各セルの valuetype を調べているのだ。値型が t から始まるときは、そのセルの textvalueformat 属性を見て生成方法を判断する。n から始まるときは、そのかわりに nontextvalueformat 属性を利用する。

しかし、セルの textvalueformat あるいは nontextvalueformat が明示的に定義されていない場合は、デフォルトの書式を valuetype から調べる。その様子は図 19.13 のようになる。

図 19.13: 値型

text-wiki の値の書式に対応するコードは SocialCalc.format_text_for_display にある。

```
if (SocialCalc.Callbacks.expand_wiki && /^text-wiki/.test(valueformat)) {
    // do general wiki markup
    displayvalue = SocialCalc.Callbacks.expand_wiki(
        displayvalue, sheetobj, linkstyle, valueformat
    );
}
```

format_text_for_display の中でインラインで Wiki から HTML に展開するのではなく、新たなフックを SocialCalc.Callbacks に定義した。これは、SocialCalc のコードベース全体を通して推奨されている方式である。Wiki 記法の展開を別の方法に差し替えるのも簡単になる。また、SocialCalc を組み込んだアプリケーション側も、その機能が不要であっても互換性を保てる。

Wikitext のレンダリング

次に、Wikiwyg[1] を利用する。これは JavaScript のライブラリで、wikitext と HTML の相互変換を行うものだ。

私たちは expand_wiki 関数を用意した。この関数はセルのテキストを受け取って、それを Wikiwyg の wikitext パーサーと HTML エミッターに渡す。

```
var parser = new Document.Parser.Wikitext();
var emitter = new Document.Emitter.HTML();
SocialCalc.Callbacks.expand_wiki = function(val) {
    // val を Wikitext から HTML に変換する
    return parser.parse(val, emitter);
}
```

[1] https://github.com/audreyt/wikiwyg-js

最後のステップは、スプレッドシートの初期化直後に `set sheet defaulttextvalueformat text-wiki` コマンドを実行させるようにすることだ。

```
// 既に <div id="tableeditor"/> が DOM に存在するものとする
var spreadsheet = new SocialCalc.SpreadsheetControl();
spreadsheet.InitializeSpreadsheetControl("tableeditor", 0, 0, 0);
spreadsheet.ExecuteCommand('set sheet defaulttextvalueformat text-wiki');
```

これらをまとめると、Render ステップでの動きは図 19.14 のようになる。

図 19.14: レンダリング手順

これで完了！進化した SocialCalc は、今や Wiki 記法にも対応するようになった。

```
*bold* _italic_ `monospace` {{unformatted}}
> indented text
* unordered list
# ordered list
"Hyperlink with label"<http://softwaregarden.com/>
{image: http://www.socialtext.com/images/logo.png}
```

試しに、A1 に`*bold* _italic_ `monospace``と入力してみよう。図 19.15 のような感じでリッチテキストに変換されるはずだ。

19.7 リアルタイムのコラボレーション

次に紹介する例は、共有スプレッドシートにおける複数ユーザーでの同時編集だ。一見複雑に思えるが、SocialCalc のモジュラー設計のおかげでそれほどでもなかった。必要なのは、各オンラインユーザーが自分のコマンドを他のメンバーにブロードキャストすることだけだ。

図 19.15: Wikywyg の例

ローカルで発行したコマンドとリモートコマンドを区別するために、ScheduleSheetCommands メソッドに isRemote パラメータを追加した。

```
SocialCalc.ScheduleSheetCommands = function(sheet, cmdstr, saveundo, isRemote) {
   if (SocialCalc.Callbacks.broadcast && !isRemote) {
       SocialCalc.Callbacks.broadcast('execute', {
           cmdstr: cmdstr, saveundo: saveundo
       });
   }
   // ... 元からあった ScheduleSheetCommands のコードが続く...
}
```

つまり、必要なのは、適切なコールバック関数 SocialCalc.Callbacks.broadcast を定義することだけだ。それさえできれば、同じスプレッドシートに接続するすべてのユーザーが同じコマンドを実行することになる。

この機能を OLPC(One Laptop Per Child[2]) 用にはじめて実装したのは SEETA の Sugar Labs[3] で、それは 2009 年のことだった。このときの broadcast 関数の実装は、D-Bus/Telepathy への XPCOM 呼び出しだった。D-Bus/Telepathy は OLPC/Sugar ネットワークの標準のトランスポートである (図 19.16 を参照)。

これはそれなりにうまく機能して、同じ Sugar ネットワーク上にある XO のインスタンスが同じ SocialCalc スプレッドシートを使って共同作業できるようになった。しかしこの方式は Mozilla/XPCOM のプラットフォームでしか使えないし、D-Bus/Telepathy メッセージングプラットフォームでしか使えない。

クロスブラウザのトランスポート

別のブラウザや別の OS でも機能するようにするために、Web::Hippie[4] フレームワークを採用した。これは JSON-over-WebSocket を上位レベルで抽象化したものに便利な jQuery バインディングを追加したものである。そして、WebSocket が使えないときに使う代替の撮らん図ポートとして MXHR (Multipart XML HTTP Request[5]) を用意した。

---

[2] http://one.laptop.org/
[3] http://seeta.in/wiki/index.php?title=Collaboration_in_SocialCalc
[4] http://search.cpan.org/dist/Web-Hippie/
[5] http://about.digg.com/blog/duistream-and-mxhr

図 19.16: OLPC の実装

Adobe Flash プラグインは入っているけれども WebSocket にはネイティブで対応していないというブラウザ向けには、web_socket.js[6] プロジェクトが作った Flash による WebSocket エミュレーションを使った。たいていの場合、こちらのほうが MXHR よりも高速だし安定していた。処理フローは図 19.17 のようになる。

[6] https://github.com/gimite/web-socket-js

図 19.17: クロスブラウザのフロー

クライアント側の SocialCalc.Callbacks.broadcast 関数の定義は、このようになる。

```
var hpipe = new Hippie.Pipe();

SocialCalc.Callbacks.broadcast = function(type, data) {
    hpipe.send({ type: type, data: data });
};

$(hpipe).bind("message.execute", function (e, d) {
    var sheet = SocialCalc.CurrentSpreadsheetControlObject.context.sheetobj;
    sheet.ScheduleSheetCommands(
        d.data.cmdstr, d.data.saveundo, true // isRemote = true
    );
    break;
});
```

これは非常にうまく機能したが、実はまだ二つの問題が残っていた。

衝突の解決

最初の問題は、コマンドを順に実行するときに発生するレースコンディションだ。あるユーザー A と別のユーザー B が、同じセルを同時に操作したとする。お互いが実行したコマンド

は相手にもブロードキャストされ、図 19.18 に示すように最終的にはお互いの状態が異なってしまう。

図 19.18: レースコンディションの衝突

この問題を解決するために使ったのが SocialCalc に組み込まれている undo/redo の仕組みだ。図 19.19 に示すように利用した。

衝突を回避するための手順は次のとおりだ。あるクライアントがコマンドをブロードキャストするときには、そのコマンドをペンディングキューに追加する。あるクライアントがコマンドを受信するときには、リモートコマンドをペンディングキューと比べてチェックする。

ペンディングキューが空の場合は、受信したコマンドをリモートアクションとしてシンプルに実行する。リモートコマンドと同じコマンドがペンディングキューにあった場合は、ローカルコマンドをキューから取り除く。

それ以外の場合は、キューに入っているコマンドの中で今受信したコマンドと衝突するものがあるかどうかを調べる。衝突するコマンドが見つかったら、クライアントはまずそのコマンドを Undo して、後で Redo するよう印をつけておく。衝突するコマンドをいったん取り消せば、リモートコマンドをいつもどおり実行できる。

Redo マークがついたコマンドをサーバーから受信した場合は、クライアントはそれをもう一度実行する。そしてキューから削除する。

図 19.19: レースコンディションの衝突の解決

リモートカーソル

レースコンディションの問題が解決できたとしても、別のユーザーがいま編集中のセルを不意に上書きしてしまう問題を完全に防げるわけではない。単純な改良方法としては、すべてのクライアントが自分のカーソルの位置を他のユーザーにブロードキャストするというものがある。そうすれば、誰が今どのセルを編集しているのかを誰もが見られるようになる。

このアイデアを実装するために、MoveECellCallback イベントに別の broadcast ハンドラを追加した。

```
editor.MoveECellCallback.broadcast = function(e) {
    hpipe.send({
        type: 'ecell',
        data: e.ecell.coord
    });
};

$(hpipe).bind("message.ecell", function (e, d) {
    var cr = SocialCalc.coordToCr(d.data);
    var cell = SocialCalc.GetEditorCellElement(editor, cr.row, cr.col);
    // ... そこを編集しているリモートユーザーがわかるようにセルの書式を変える...
});
```

スプレッドシート上で作業中のセルに印をつける方法として一般的なのは、色つきの枠線で囲むことだ。しかし、もともとそのセルには `border` プロパティを設定してある可能性もある。また、`border` は単色なので、これだとひとつのセルの上に複数のカーソルがある場合に対応できない。

そこで、CSS3 に対応しているブラウザ上では `box-shadow` プロパティを使って同一セル上の複数のカーソルを表すようにした。

```
/* ふたつのカーソルが同一セル上にある場合 */
box-shadow: inset 0 0 0 4px red, inset 0 0 0 2px green;
```

図 19.20 は、四人で同じスプレッドシートを編集したらどうなるかを示す図だ。

図 19.20: 四人のユーザーによる同時編集

19.8 教訓

SocialCalc 1.0 を公開したのは 2009 年 10 月 19 日。VisiCalc の最初のリリースからちょうど 30 年後のことだった。Socialtext 社の同僚とともに Dan Bricklin の指導のもとで作業したという経験はとても有益なものだった。そのときに学んだ教訓を、ここでみなさんと共有したい。

明確なビジョンを持ったチーフデザイナー

[Bro10] において、Fred Brooks は複雑なシステムを構築するときのことについて述べている。一貫した**設計コンセプト**に焦点を合わせるほうが、そこから派生したさまざまな表現を見るよりもより対話が直接的になるということだ。Brooks によると、一貫した設計コンセプトは誰かひとりで作り上げるのが一番だという。

> 大規模な設計では概念の整合性が最も重要になるし、そのためには一人あるいはごく少数の人たちが**心をひとつ** *(uno animo)* にしなければならないので、よくわかっているマネージャーは才能があるチーフデザイナーに設計を任せることになる。

SocialCalc の場合は、Tracy Ruggles をチーフ UX デザイナーとして迎えたことでプロジェクトのビジョンを共有できるようになった。元になる SocialCalc のエンジンはとても素直な作りだったので、場当たり式の機能追加で使いにくいものになってしまう危険性はかなり高かった。Tracy がデザインスケッチを使って話してくれたおかげで、ユーザーの直感に反するような機能を作り込んでしまわずに済んだ。

プロジェクトの継続性のための Wiki

私が SocialCalc プロジェクトに合流するまでに、既に二年以上かけて設計や開発が進んでいた。しかし、これまでの流れを把握して実際に作業を始めるまでには一週間もかからなかった。というのも、**すべては *Wiki* に書かれていた**からである。最初期の設計メモもあれば最新版のブラウザ対応表もあって、これまでの流れがすべて Wiki ページと SocialCalc のスプレッドシートにまとまっていたのだ。

プロジェクトのワークスペースを読めば、すぐにその内容が頭に入るようになった。よくありがちな「新入りに誰か一人をつけてオリエンテーションをする」ような手間は不要だった。

こんなこと、昔ながらのオープンソースプロジェクトではできなかっただろう。かつてはほとんどのやりとりを IRC やメーリングリストで済ませていたし、仮に Wiki があったとしてもそこにあるのはドキュメントや開発リソースへのリンクだけだった。新入りがチームの仲間入りをするには、体系化されていない IRC の過去ログやメーリングリストのアーカイブを読んで背景を共有する必要があった。

タイムゾーンの違いを受け入れる

Ruby on Rails の作者である David Heinemeier Hansson は、37signals に合流したときのことを引き合いに出して分散チームの利点を説いたことがある。「コペンハーゲンとシカゴの間には 7 つのタイムゾーンがある。つまり、変な横やりが入る心配をせずに多くの作業をこなせるってことだ」。台北とパロアルトの間には 9 つのタイムゾーンがある。私たちが SocialCalc の開発をしているときにも同じように感じた。

設計-開発-QA のフィードバックサイクルを 24 時間以内で回せることも多かった。各フェーズを、それぞれのタイムゾーンにおける昼間の 8 時間で済ませたのだ。このような非同期型の共同作業を進めるには、自己言及的な (それだけで意味がわかる) 成果物 (デザインスケッチやコード、そしてテスト) を作らざるを得なくなる。それがまた、お互いの信頼関係を高めることにつながった。

楽しくやろうよ

2006 の CONISLI カンファレンス [Tan06] のキーノートで私は、Perl 6 の実装を進める分散チームを率いた経験をまとめていくつかの所感を述べた。その中でも *Always have a Roadmap(* **常**

にロードマップを)、*Forgiveness > Permission(***許可を求めるのではなくやってから許しを請え**)、*Remove deadlocks(***デッドロックをなくせ**)、*Seek ideas, not consensus(***アイデアを探せ。合意を得ようとするな**)、そして *Sketch ideas with code(***アイデアのスケッチにコードをつけよ**) あたりは小規模な分散チームに特にかかわってくるものだ。

SocialCalc の開発にあたって私たちが重視したのは、知識をチーム全体に広めること。コードを共同所有する方式にしたので、誰かひとりがクリティカルなボトルネックになることがなくなった。

さらに、不和を事前に回避するために、実際に代替案のコードを書いてその設計を検証したりした。そして、よりよい設計が見つかった場合は、きちんと動いているプロトタイプがあったとしても躊躇せず新たな設計に書き換えた。

こういった文化があったおかげで、先を予想する力がついたし、実際に顔を合わせて対話することがなくてもチームの仲間意識は育っていった。そしてチーム運営に関する面倒なことは最小限に抑えて SocialCalc の作業を楽しいものにしたのだ。

ストーリーテスト駆動開発

Socialtext 社に合流するまでの私は「テストと仕様策定を交互に行う」方式を支持していた。Perl 6 の仕様[7]にも見られるように、言語仕様の注記として公式のテストスイートをつけていたのだ。しかし、SocialCalc の QA チームにいた Ken Pier と Matt Heusser にその考えを改めさせられた。もう一歩先に進んで、テスト自体を**実行可能な仕様**にできると知ったのだ。

[GR09] の第 16 章で、Matt は私たちのストーリーテスト駆動開発についてこのように述べている。

> 作業の基本単位は「ストーリー」という、きわめて小さな要件文書です。ストーリーにはフィーチャの簡単な説明と、ストーリーの完成時に何を検討すべきかについての実例が含まれています。これらの実例は「受け入れテスト」と呼ばれ、平易な英語で記述します。
>
> ストーリーの開始では、まずプロダクトオーナーが叩き台としての受け入れテストを作成し、それを開発者とテスターが受け取ってから、初めて開発者がコーディングを開始します。

このストーリーテストを wikitest に変換する。これはテーブルベースの仕様記述言語で、Ward Cunningham の FIT フレームワーク[8]を参考にしたものだ。wikitest は、`Test::WWW::Mechanize`[9] や `Test::WWW::Selenium`[10] といったテスティングフレームワークの自動化を行う。

ストーリーテストを普通の言葉で書き表して要件を検証する。そのメリットは計り知れない。誤解を減らす助けにもなるし、毎月のリリースでの手戻りがほとんど発生しなくなった。

[7] `http://perlcabal.org/syn/S02.html`
[8] `http://fit.c2.com/`
[9] `http://search.cpan.org/dist/Test-WWW-Mechanize/`
[10] `http://search.cpan.org/dist/Test-WWW-Selenium/`

オープンソースと CPAL

あとひとつ言い忘れたことが。SocialCalc に適用したオープンソースモデルそれ自体も興味深い教訓になった。

Socialtext 社は SocialCalc 用に新たなライセンス Common Public Attribution License[11] を作った。Mozilla Public License を元にして作った CPAL は、ソフトウェアのユーザーインターフェイス上にクレジットを入れることを原作者が強要できるようにしている。また、ネットワーク上での利用を考慮した条項もあって、派生物をネットワーク上でホストする場合に同じ条件で配布することを要求する。

Open Source Initiative[12] と Free Software Foundation[13] の両方の認証を得たことで、Facebook[14] や Reddit[15] といった人気サイトがそのプラットフォームのソースコードを CPAL でリリースするようになった。これはとても励みになった。

CPAL は "弱いコピーレフト" なライセンスなので、他のフリーソフトウェアやプロプライエタリなソフトウェアとも自由に組み合わせることができる。唯一必要なのは、SocialCalc 自体に手を入れたらそれを公開することだけだ。このおかげでさまざまなコミュニティから SocialCalc への協力を得られ、よりすばらしいものになった。

このオープンソースのスプレッドシートエンジンにはまだまだ可能性に満ちあふれている。もしあなたのお気に入りのプロジェクトに SocialCalc を組み込むことができたら、ぜひ私たちにも教えて欲しい。

[11] https://www.socialtext.net/open/?cpal
[12] http://opensource.org/
[13] http://www.fsf.org
[14] https://github.com/facebook/platform
[15] https://github.com/reddit/reddit

第20章

Telepathy

Danielle Madeley

Telepathy[1] はリアルタイム通信のためのモジュラーフレームワークで、音声や動画、テキスト、ファイル転送などを扱える。Telepathy が他のフレームワークと違う点は、さまざまなインスタントメッセージングプロトコルの詳細を抽象化しているというところではない。サービスとしての通信 (communications as a service) を提供するというアイデアこそが他との違いで、これはちょうど印刷をサービスとして提供して多くのアプリケーションから使えるようにするという考え方と同じである。このアイデアを実現するために、Telepathy は D-Bus メッセージングバスとモジュラー設計を幅広く活用した。

通信をサービスとして提供できると非常に便利だ。単一のアプリケーションの枠を超えた通信ができるようになるからだ。いろんな使い道が考えられる。たとえば、メールアプリケーション上で相手が在席中であることを確認してから相手とのやりとりを始めたり、ファイルを転送も、ファイルブラウザから相手に直接送れたりする。アプリケーション内でお互いに共同作業することもできる。これは Telepathy as *Tubes* (Telepathy を土管として使う) と呼ばれている。

Telepathy は Robert McQueen が 2005 年に作ったものだ。それ以降はいくつかの企業や個人で開発と保守を進めている。その中の一社である Collabora は、McQueen が共同設立者の一人となっている企業である。

20.1 Telepathy Framework のコンポーネント

Telepathy はモジュラー構造で、各モジュール間の通信には D-Bus メッセージングバスを使う。たいていの場合は、ユーザーのセッションバスである。通信の詳細は Telepathy の仕様[2]を参照して欲しい。Telepathy フレームワークのコンポーネント群を図 20.1 に示す。

[1] `http://telepathy.freedesktop.org/`、あるいは `http://telepathy.freedesktop.org/doc/book/` にある開発者向けマニュアルを参照

[2] `http://telepathy.freedesktop.org/spec/`

D-Bus メッセージバス

D-Bus はプロセス間通信用の非同期メッセージバスで、GNOME や KDE といったデスクトップ環境を含むほとんどの GNU/Linux システムのバックボーンとなっている。D-Bus はそもそも共有バスアーキテクチャだった。アプリケーションはソケットアドレスを指定してバスに接続し、ターゲットメッセージをバス上の別のアプリケーションに送信したりバス上の全メンバーにシグナルをブロードキャストしたりできる。バス上のアプリケーションは、IP アドレスと同じような感じのバスアドレスを持つ。また、名前を付けることもできる。DNS と同じような感じで、たとえば `org.freedesktop.Telepathy.AccountManager` のようになる。すべてのプロセスが D-Bus デーモン経由で通信を行う。このデーモンが、メッセージの受け渡しや名前の登録を処理する。

ユーザーの視点で見ると、すべてのシステムにはふたつのバスが存在する。ひとつはシステムバスで、これはユーザーがシステム全体のコンポーネント (プリンタや Bluetooth、ハードウェア管理など) と通信できるようにするためのバスである。システム上のすべてのユーザーが共有する。もうひとつはセッションバスで、これは各ユーザーに固有のものである (つまり、システムにログインしているユーザーの数だけセッションバスが存在する)。これは、そのユーザーが使うアプリケーションどうしがお互いに通信するために使う。セッションバス上に大量のトラフィックが流れている場合は、アプリケーションが自前のプライベートバスを作ることもできる。あるいは、`dbus-daemon` を介さないピアツーピアのバスを作ることもできる。

D-Bus プロトコルを実装して D-Bus デーモンと通信できるようにしたライブラリがいくつか存在する。libdbus や GDBus、QtDBus、そして python-dbus などだ。これらのライブラリの役割は、D-Bus メッセージの送受信だけでなく各言語の型システムから D-Bus の型フォーマットへの変換やバス上のオブジェクトの公開などもある。これらのライブラリは、便利な API を提供していることも多い。接続中のアプリケーションの一覧やアクティベート可能なアプリケーションの一覧を表示したり、バス上での名前を取得したりするための API である。D-Bus レベルでは、これらすべての操作は `dbus-daemon` 自身が公開するオブジェクト上でのメソッド呼び出しで行う。

D-Bus についての詳細は `http://www.freedesktop.org/wiki/Software/dbus` を参照してほしい。

- Connection Manager は、Telepathy と個々の通信サービスとの間のインターフェイスを提供する。たとえば XMPP 用の Connection Manager もあれば SIP 用や IRC 用などの Connection Manager もあるという具合だ。Telepathy で新たなプロトコルに対応しよう

図 20.1: Telepathy コンポーネントの例

と思ったら、単にそのプロトコル用の Connection Manager を書けばよい。

- Account Manager は、ユーザーの通信用アカウントを格納して各クライアントとの接続を確立する。その際に、リクエストがあれば適切な Connection Manager を利用する。
- Channel Dispatcher の役割は、各 Connection Manager からの入力チャネルをリスンしてチャネルの型 (テキストや音声、動画、ファイル転送、チューブなど) に応じてそれを処理できるクライアントに振り分ける。Channel Dispatcher は、それ以外にもサービスを提供する。さまざまなアプリケーション、特に Telepathy クライアントではないアプリケーションから出力チャネルにリクエストできるようにして、それをローカルの適切なクライアントで処理させるサービスだ。これを使えば、メールソフトなどのアプリケーションがテキストチャットのリクエストを出せるようになり、リクエストがあれば、IM クライアントがチャットウィンドウを表示することになる。
- Telepathy クライアントは、通信チャネルを処理したり監視したりする。IM や VoIP クライアントのようなユーザーインターフェイスだけでなく、チャットのロガーのようなサービスも含む。クライアントは自分自身を Channel Dispatcher に登録し、自分が処理したいチャネルタイプを伝える。

現在の Telepathy の実装では、Account Manager と Channel Dispatcher はひとつのプロセスで提供している。このプロセスのことを Mission Control と呼ぶ。

このモジュラー設計は、Doug McIlroy の哲学「一つのことを行い、またそれをうまくやるプログラムを書け」に基づいたもので、次のような利点がある。

ロバストネス:どれかひとつのコンポーネントに障害が発生しても、サービス全体がクラッシュすることはない。
開発のしやすさ:稼働中のシステムのコンポーネントを差し替えたとしても、他の箇所には影響しない。開発版のモジュールを、動作検証済みの他のモジュールと組み合わせてテストすることもできる。
言語非依存:コンポーネントは、D-Bus バインディングがありさえすればどんな言語でも書ける。もし何かの通信プロトコル用の実装としてすぐれたものが何かの言語で用意されているのなら、Connection Manager をその言語で書けばよい。他の言語で書いたものも含め、すべての Telepathy クライアントからそれを使うことができる。同様に、何か特定の言語でユーザーインターフェイスを作れば、そこからすべてのプロトコルにアクセスできる。
ライセンス非依存:自分で作ったコンポーネントを、まったく別のライセンスで公開することもできる。すべてがひとつのプロセスの場合だと非互換になってしまうようなライセンスでも、だいじょうぶだ。
インターフェイス非依存:同じ Telepathy コンポーネントに対して複数のユーザーインターフェイスを開発できる。デスクトップ環境やハードウェアデバイス (GNOME や KDE、Meego、Sugar など) ごとにネイティブインターフェイスを用意できるということだ。
セキュリティ:コンポーネントは個別のアドレス空間で動作し、限られた権限しか持たない。たとえば、一般的な Connection Manager が必要な権限はネットワークへのアクセスと D-Bus セッションバスへのアクセスだけである。それを実現するために、SELinux 風の仕組みを使ってそのコンポーネントがアクセスできる範囲を制限している。

Connection Manager はコネクションを管理し、各コネクションは通信サービスへの論理的な接続を表す。設定済みのアカウントごとにひとつのコネクションが存在する。ひとつのコネクションには複数のチャネルが含まれる。チャネルとは、通信を運ぶ仕組みである。チャネルは、IM の会話や音声通信、映像通信、ファイル転送などのステートフルな操作となる。コネクションとチャネルの詳細については 20.3 節で説明する。

20.2 Telepathy による D-Bus の利用法

Telepathy コンポーネントは D-Bus メッセージングバスを使って通信する。通常はユーザーのセッションバスだ。D-Bus が提供する機能は、他のプロセス間通信システムにもよくあるものだ。各サービスはオブジェクトを公開し、それぞれがきちんと名前空間に分けられたオブジェクトパスを持つ。たとえば`/org/freedesktop/Telepathy/AccountManager`[3] のようなものだ。各オブジェクトはさまざまなインターフェイスを実装する。これもまたきちんと名前空間に分かれており、`org.freedesktop.DBus.Properties` や `ofdT.Connection` のような形式になる。それぞれのインターフェイスが、メソッドやシグナルそしてプロパティを提供する。これらを呼んだりリスンしたりリクエストしたりするわけだ。

図 20.2: D-Bus サービスが公開するオブジェクトの概念表現

D-Bus オブジェクトの公開

D-Bus オブジェクトの公開は、使っている D-Bus ライブラリが全体を担当する。実際は、D-Bus オブジェクトのパスとそのインターフェイスを実装するソフトウェアオブジェクトとのマッピングである。サービスが公開するオブジェクトのパスは、オプションの `org.freedesktop.DBus.Introspectable` インターフェイスで公開する。

あるサービスが、指定したパス (`/ofdT/AccountManager` など) からのメソッド呼び出しを受け取ると、D-Bus ライブラリがその D-Bus オブジェクトを提供するソフトウェアオブジェクトの場所を探し、それに対して適切なメソッド呼び出しを行う。

Telepathy が提供するインターフェイスやメソッド、シグナル、そしてプロパティは XML ベースの D-Bus IDL で詳述される。それだけでなく、ここにはより詳細な情報も含まれている。この仕様をパースして、ドキュメントを生成したり言語バインディングを生成したりできる。

Telepathy サービスは、さまざまなオブジェクトをバス上に公開する。Mission Control が公開するのは Account Manager や Channel Dispatcher 用のオブジェクトで、これを使えばそれぞれのサービスにアクセスできるようになる。クライアントが公開するのは Client オブジェク

[3] これ以降は、`/org/freedesktop/Telepathy/`や `org.freedesktop.Telepathy` を `ofdT` と略記する。

トで、これを使えばChannel Dispatcherからクライアントにアクセスできるようになる。最後に、Connection Managersもいくつかのオブジェクトを公開する。Account Managerが新たなコネクションを要求するために使うサービスオブジェクトや各コネクションに対応するオブジェクト、そしてチャネルに対応するオブジェクトなどだ。

D-Busバスオブジェクトは型を持たない(インターフェイスしか持たない)が、Telepathyはいくつかの方法で型をシミュレートする。オブジェクトのパスを見れば、そのオブジェクトがコネクションなのかチャネルなのかクライアントなのかといったことはわかる。しかし普通は、プロキシにオブジェクトをリクエストした時点で既にそのパスを知っているはずだ。各オブジェクトは、その型に合わせた基底インターフェイスを実装している。`ofdT.Connection`や`ofdT.Channel`といったものだ。チャネルにとっては、これは一種の抽象基底クラスのようなものになる。Channelオブジェクトはそのチャネルの型を定義する具象クラスを持つことになる。改めて言うが、これはD-Busのインターフェイスで表されるものだ。チャネルの型について知るには、Channelインターフェイスの`ChannelType`プロパティを読めばよい。

最後にもうひとつ。各オブジェクトはオプションのインターフェイスも実装している(言うまでもなく、これらもまたD-Busのインターフェイスだ)。これらは、そのプロトコルやConnection Managerの機能に依存する。あるオブジェクトで使えるインターフェイスを知るには、そのオブジェクトの基底クラスの`Interfaces`プロパティを調べればよい。

`ofdT.Connection`型のConnectionオブジェクトの場合、オプションのインターフェイスの名前は`ofdT.Connection.Interface.Avatars`(アバターの概念を持つプロトコルの場合)や`odfT.Connection.Interface.ContactList`(連絡先名簿を提供するプロトコルの場合—そうでないものもある)そして`odfT.Connection.Interface.Location`(位置情報を提供するプロ所ルの場合)となる。`ofdT.Channel`型のChannelオブジェクトの場合は、具象クラスが持つインターフェイスの名前は`ofdT.Channel.Type.Text`や`odfT.Channel.Type.Call`そして`odfT.Channel.Type.FileTransfer`といった形式になる。コネクションの場合と同様、オプションのインターフェイスの名前は`odfT.Channel.Interface.Messages`(このチャネルでテキストメッセージの送受信ができる場合)や`odfT.Channel.Interface.Group`(このチャネルの相手が複数の連絡先を含むグループ、たとえばマルチユーザーチャットである場合)のようになる。つまり、たとえばテキストチャネルの場合は、少なくとも`ofdT.Channel`と`ofdT.Channel.Type.Text`そして`Channel.Interface.Messages`インターフェイスを実装していることになる。もしそのチャネルがマルチユーザーチャットであるなら、さらに`odfT.Channel.Interface.Group`も実装しているわけだ。

D-Bus自体には、コネクションオブジェクトがコネクション関係のインターフェイスだけしか持っていないことを確認する手段がない(というのもD-Busには型という概念がなく、任意の名前のインターフェイスがあるだけだからである)。しかし、Telepathyの仕様に含まれる情報を使えば、Telepathyの言語バインディングの中でそのチェックはできる。

D-Bus イントロスペクションがあるのになぜ Interfaces プロパティなのか

なぜわざわざ各基底クラスが `Interfaces` プロパティを実装しているのだろう？D-Bus のイントロスペクション機能を使えばどんなインターフェイスがあるかはわかるはずなのに。その理由を説明する。さまざまなチャネルやコネクションオブジェクトはそれぞれ異なるインターフェイスを持つ可能性があり、それはチャネルやコネクションの機能に依存している。しかし D-Bus のイントロスペクション機能の大半は、同じクラスのオブジェクトはすべて同じインターフェイスを持つことを前提としている。たとえば `telepathy-glib` では、D-Bus のイントロスペクションが列挙する D-Bus インターフェイスはそのクラスが実装するオブジェクトのインターフェイスから取得したものであり、コンパイル時に静的に決まるものである。私たちはこの問題を解決するために、D-Bus のイントロスペクションがオブジェクトに存在しないものも含めてすべてのインターフェイスのデータを提供させるようにした。そして、`Interfaces` プロパティを使って実際に動くものがどれなのかを示すようにした。

ハンドル

ハンドルは、Telepathy の中での識別子 (連絡先やルーム名など) として利用される。Connection Manager が割り当てる符号なし整数値で、タプル (connection, handle type, handle) で連絡先やルームを一意に特定できる。

なぜ、そしてどうやって Specification IDL を展開するのか

既存の D-Bus specification IDL で定義されているのは名前や引数そしてアクセス制御の他にはメソッドやプロパティそしてシグナルの D-Bus 型シグネチャである。ドキュメンテーションやバインドヒントあるいは型の名前などには対応していない。この制約を解決するために、新たな XML 名前空間を追加して必要な情報を提供することにした。この名前空間は汎用的に作られている。他の D-Bus API からも使えるようにするためだ。新たに追加された要素を使えば、インラインドキュメントや論理的な意味、簡単な解説、廃止予定のバージョン、発生しうる例外などといった情報をメソッドに含められるようになる。

D-Bus の型シグネチャは低レベルの型記法であり、バス上で何がシリアライズされるのかを表している。D-Bus の型シグネチャは `(ii)` (これは、ふたつの int32 を含む構造体を表す) のようなものだったり、あるいはもう少し複雑なものだったりする。たとえば `a{sa(usuu)}`は文字列から構造体の配列へのマップで、その構造には uint32、string、uint32 そして uint32 が含まれる (図 20.3)。これらの型情報にはデータフォーマットについての説明しか含まれておらず、その型に含まれる情報の意味はまったくわからない。

プログラマーにとっての意味を明確にして言語バインディングの型付けを強化するために、新たな要素が追加された。単純な型や構造体、マップ、列挙型、フラグなどについて、その型シグネチャだけでなくドキュメンテーションも提供できるようにしたのだ。また、D-Bus オブジェクトでのオブジェクト継承をシミュレートするための要素も追加された。

通信プロトコルによって識別子の正規化の方法が違う (大文字小文字の区別やリソースの扱いなど) ので、クライアント側からふたつの識別子が一致するかどうかを知る方法をハンドルが提供する。異なるふたつの識別子のハンドルをリクエストして、もし両方のハンドル番号が一致していればそのふたつの識別子は同じ連絡先あるいはルームを指していることになる。

プロトコルによって識別子の正規化ルールは違うのだから、クライアント側で識別子の文字列を使った比較をしても無意味だ。たとえば`escher@tuxedo.cat/bed` と `escher@tuxedo.cat/litterbox` は、XMPP プロトコルではどちらも同じ連絡先 (`escher@tuxedo.cat`) を表すふたつのインスタンスとなる。つまりこれらふたつのハンドルは一致する。クライアント側からは、チャネルに対して識別子を要求することもできるしハンドルを要求することもできる。しかし、比較に使えるのはハンドルだけであることに注意。

図 20.3: D-Bus の (ii) 型および a{sa(usuu)}型

Telepathy サービスの検出

Account Manager や Channel Dispatcher など、サービスによっては常に存在するものもあり、これらについては Telepathy の仕様で名前が定義されている。しかし Connection Managers やクライアントには既知の名前がないので、使うときにはサービスを探す必要がある。

Telepathy には、稼働中の Connection Managers やクライアントの登録を受け持つサービスはない。その代わりに、D-Bus 上に新たなサービスが現れたときのアナウンスを聞いて調べることになる。D-Bus バスデーモンは、新たな D-Bus サービスがバス上に登場するたびにシグナルを発行する。クライアントや Connection Managers の名前は使用で定義されたプレフィックスから始まるので、新たな名前をこのプレフィックスとマッチさせて調べることができる。

この方式のメリットは、完全にステートレスであることだ。Telepathy コンポーネントが立ち上がるときに、バスデーモン (コネクションが開いているときには、正式な一覧がある) に対してどんなサービスが稼働中なのかを問い合わせることができる。たとえば仮に Account Manager がクラッシュした場合、いまどんなコネクションが稼働中なのかを確認してそれをアカウントオブジェクトと再びつなげることができる。

Channel Dispatcher もこの方式を使って Telepathy クライアントを探す。クライアントの名前は `ofdT.Client` ではじまり、たとえば `ofdT.Client.Logger` のようになる。

コネクションもまたサービス

Connection Managers 自身と同様、コネクションもまた D-Bus のサービスとなる。この前提だと、Connection Manager は各コネクションを個別のプロセスとしてフォークできるようになる。しかし現時点では、そんな実装の Connection Manager は存在しない。より現実的な手段として、すべての稼働中のコネクションを調べるときには D-Bus バスデーモンに対して `ofdT.Connection` ではじまるすべてのサービスを問い合わせる。

D-Bus トラフィックの軽減

最初の Telepathy の仕様は、大量の D-Bus トラフィックが発生するものだった。メソッドを呼ぼうとすると、バス上の大量のコンシューマーが必要とする情報をリクエストすることになったのだ。その後のバージョンで、Telepathy はこれに対処するためにさまざまな最適化を施した。

個々のメソッド呼び出しは D-Bus のプロパティに置き換えられた。当初の仕様では、オブジェクトのプロパティごとに `GetInterfaces` や `GetChannelType` といった個別のメソッドがあったのだ。オブジェクトのすべてのプロパティを取得したければメソッド呼び出しが何回も発生し、そのたびに呼び出しのオーバーヘッドが発生することになる。D-Bus のプロパティを使うことで、すべてのプロパティを取得するには標準の `GetAll` メソッドを一度使うだけでいいことになった。

さらに、チャネル上のプロパティの大半は、そのチャネルの活動中は不変なものだ。チャネルの型やインターフェイス、接続先、リクエスト元などがその一例である。たとえばファイル転送チャネルなら、それ以外にもファイルサイズやコンテントタイプなども含まれる。

チャネル (入力側もリクエストの送出側も) の作成を事前に通知するシグナルが追加され、不変なプロパティ用のハッシュテーブルもそこに含むようにした。これはチャネルプロキシのコンストラクタ (20.4 節を参照) に直接渡せて、これを使えば各クライアントが個別に情報をリクエストせずに済むようになる。

ユーザーのアバターがバス上を通るときには、バイト配列形式になる。Telepathy は既にアバターを参照するトークンを使っており、それを見れば新たなアバターをダウンロードする必要があるかどうかを判断できた。不要なダウンロードの手間はこれで省けていたが、アバターをダウンロードするときには各クライアントが個別に `RequestAvatar` メソッドを呼ぶ必要があった。このメソッドは、アバターを返すものだ。つまり、ある連絡先がアバターを更新したと Connection Manager が通知すると、そのアバターを取得しようとするリクエストがあちこちから別々に発生し、メッセージバス上を同じアバターが複数回転送されることになっていた。

これを解決するために新たなメソッドを追加した。このメソッドはアバターを返さない (何も返さない)。そのかわりに、アバターをリクエストキューに追加する。アバターをネット

ワークから取得すると AvatarRetrieved シグナルが発生する。各クライアントは、このシグナルを待ち受けることができる。これはつまり、アバターのデータがバス上で転送されるのは一度だけで済むということだ。それ以降は、すべてのクライアントがそのアバターを使えるようになる。あるクライアントのリクエストがキューに入れば、それ以降のクライアントからのリクエストは AvatarRetrieved が発行されるまでは無視される。

たくさんの連絡先を読み込む必要がある場合 (連絡先一覧を読み込む場合など) は、大量の情報をリクエストしなければいけない。エイリアス、アバター、所属グループ、位置、アドレス、電話番号などである。かつての Telepathy では情報グループごとに個別のメソッド呼び出しが必要で (GetAliases など大半の API は既に連絡先リストを受け取るようになっていた)、その結果、10 回近くもメソッド呼び出しが発生した。

これを解決するために Contacts インターフェイスを導入した。そのおかげで、複数のインターフェイスからの情報を一度のメソッド呼び出しで返せるようになった。Telepathy の仕様は拡張され、連絡先の属性も含むようになった。GetContactAttributes メソッドが返す名前空間付きのプロパティが、連絡先情報を取得するために使うメソッド呼び出しを覆い隠したのだ。クライアントは、GetContactAttributes を呼ぶときに連絡先一覧と知りたいインターフェイスを渡す。そして、属性と値とのマップを連絡先と関連づけたマップを取得する。

コードを見たほうが話が早いだろう。まずリクエストはこのようになる。

```
connection[CONNECTION_INTERFACE_CONTACTS].GetContactAttributes(
  [ 1, 2, 3 ], # contact handles
  [ "ofdT.Connection.Interface.Aliasing",
    "ofdT.Connection.Interface.Avatars",
    "ofdT.Connection.Interface.ContactGroups",
    "ofdT.Connection.Interface.Location"
  ],
  False # don't hold a reference to these contacts
)
```

そしてその応答はこのようになる。

```
{ 1: { 'ofdT.Connection.Interface.Aliasing/alias': 'Harvey Cat',
       'ofdT.Connection.Interface.Avatars/token': hex string,
       'ofdT.Connection.Interface.Location/location': location,
       'ofdT.Connection.Interface.ContactGroups/groups': [ 'Squid House' ],
       'ofdT.Connection/contact-id': 'harvey@nom.cat'
     },
  2: { 'ofdT.Connection.Interface.Aliasing/alias': 'Escher Cat',
       'ofdT.Connection.Interface.Avatars/token': hex string,
       'ofdT.Connection.Interface.Location/location': location,
       'ofdT.Connection.Interface.ContactGroups/groups': [],
       'ofdT.Connection/contact-id': 'escher@tuxedo.cat'
     },
  3: { 'ofdT.Connection.Interface.Aliasing/alias': 'Cami Cat',
        ...
     }
}
```

20.3　コネクション、チャネル、そしてクライアント

コネクション

コネクションを作るのは Connection Manager で、単一のプロトコル／アカウントとの接続を確立する。たとえば XMPP アカウント `escher@tuxedo.cat` および `cami@egg.cat` と接続すると二つのコネクションが作られることになり、それぞれが D-Bus オブジェクトとして表される。コネクションの準備をするのはたいていの場合 Account Manager で、現在有効なアカウントに対して行う。

コネクションは、いくつかの必須機能を提供する。接続状態の管理や監視、そしてチャネルのリクエストに欠かせないものである。そしてまた、オプションの機能も提供する。どんな機能を提供するかは、そのプロトコルの機能に依存する。これらはオプションの D-Bus インターフェイス (先ほど説明したもの) として提供され、コネクションの `Interfaces` プロパティで一覧できる。

通常は、コネクションを管理するのは Account Manager で、対応するアカウントのプロパティを使って作成する。Account Manager は各アカウントに対してユーザーの存在をそれに対応する接続と同期させ、指定したアカウント用のコネクションパスを答えることができる。

チャネル

チャネルは、実際の通信を行う仕組みである。通常は IM の会話だったり音声や映像での通話だったりファイル転送だったりするが、チャネルを使ってサーバー自身とのステートフルな通信 (チャットルームや連絡先の検索など) をさせることもできる。個々のチャネルは D-Bus オブジェクトとして表される。

チャネルは一般に複数のユーザーの間でできるものであり、その中の一人があなたとなる。通常は、ターゲット ID を持っている。これは、一対一の通信の場合は通信の相手を指し、マルチユーザーの通信 (チャットルームなど) の場合はルーム ID を指す。マルチユーザーのチャネルは `Group` インテーフェイスを公開しており、これを使えば現在チャネルに参加している連絡先を追える。

チャネルはコネクションに属し、Connection Manager からのリクエストを (通常は Channel Dispatcher 経由で) 受ける。あるいは、ネットワーク上でのイベント (チャットの受信など) に対応してコネクションがチャネルを作ることもある。そしてそれを Channel Dispatcher に渡してディスパッチさせる。

チャネルの型を定義するのが、チャネルの `ChannelType` プロパティである。このチャネルの型で必要となるフィーチャ(テキストメッセージの送受信など) やメソッド、プロパティ、シグナルの定義は適切な `Channel.Type` D-Bus インターフェイスで行われる。`Channel.Type.Text` などである。チャネルの型によっては、オプションで追加機能 (暗号化など) を実装しているものもある。これらは別のインターフェイスとして、チャネルの `Interfaces` プロパティか

ら得られる。あるユーザーをマルチユーザーのチャットルームに接続させるテキストチャネルを例にして考えよう。そのインターフェイスは表 20.1 に示すようなものとなる。

odfT.Channel	すべてのチャネルに共通するフィーチャ
odfT.Channel.Type.Text	チャネルの型。テキストチャネルに共通するフィーチャも含む
odfT.Channel.Interface.Messages	リッチテキストメッセージング
odfT.Channel.Interface.Group	このチャネルのメンバーの一覧、追跡、招待、そして承認
odfT.Channel.Interface.Room	チャットルームの件名などのプロパティの読み込みと設定

表 20.1: テキストチャネルの例

コンタクトリストチャネル**:** 失敗例

最初のバージョンの Telepathy の仕様では、連絡先リストもチャネルの一種だとしていた。サーバー側で定義された連絡先リスト (購読ユーザー一覧、配信先ユーザー一覧、ブロック済みユーザー一覧など) があって、これをコネクションからリクエストできた。リストのメンバーを取得するには Group インターフェイスを使う。ちょうどマルチユーザーのチャットと同じような仕組みだ。

当初は、チャネルを作るのは連絡先リストを取得するときの一度だけでよかったが、プロトコルによってはこれは時間のかかる処理だった。クライアント側からはいつでもチャネルをリクエストでき、準備ができしだいすぐに配送されたが、多数の連絡先を持つユーザーの場合は、時にリクエストがタイムアウトすることもあった。あるクライアントの購読/配信/ブロックの状態を知るには、三つのチャネルをチェックしなければいけなかったのだ。

連絡先グループ (友人など) もチャネルとして公開され、グループごとにひとつのチャネルになっていた。クライアント側の開発者にしてみると、これは非常に使いづらいものであることがわかった。たとえば、グループの一覧を取得しようとすれば、クライアント側で大量のコードを書く必要があったのだ。さらに、情報がチャネル経由でしか得られないので、ある連絡先のグループや購読の状態を Contacts インターフェイス経由で公開できないことになる。

そのため、どちらのチャネル型についても結局はコネクション自身のインターフェイスに置き換えた。こうすることで、連絡先情報や各連絡先の購読状況、所属グループ、グループのメンバーなどをより有用な形式でクライアント側に公開できるようになった。連絡先リストの準備ができたら、シグナルで通知する。

チャネルのリクエスト、チャネルのプロパティそしてディスパッチ

チャネルをリクエストするときには、そのチャネルに持たせたいプロパティのマップを使う。一般に、チャネルのリクエストに含まれるのはチャネルの型とターゲットのハンドル型(連絡先あるいはルーム) そしてターゲットだ。しかし、チャネルのリクエストにはそれ以外のプロパティも含めることもできる。ファイル転送ならファイル名とファイルサイズ、通話なら音声と映像のどちらを使うか、どの既存のチャネルをカンファレンスコールに含めるか、どのコンタクトサーバーから連絡先を探すか、などがその一例だ。

チャネルリクエストに含まれるプロパティは、Telepathy の仕様に定められたインターフェイスで定義されているものとなる。たとえば `ChannelType` プロパティ(表 20.2) がそのひとつである。プロパティは、その定義元のインターフェイスの名前空間で表す。チャネルリクエストに含めることのできるプロパティは、Telepathy の仕様上では *requestable* と記されている。

プロパティ	値
`ofdT.Channel.ChannelType`	`ofdT.Channel.Type.Text`
`ofdT.Channel.TargetHandleType`	`Handle_Type_Contact` (1)
`ofdT.Channel.TargetID`	`escher@tuxedo.cat`

表 20.2: チャネルリクエストの例

もう少し複雑な例を表 20.3 に示す。これはファイル転送チャネルへのリクエストだ。リクエストしているプロパティが、その定義元のインターフェイス名で指定されているところに注目しよう (簡潔にするために、必須プロパティの一部を省略した)。

プロパティ	値
`ofdT.Channel.ChannelType`	`ofdT.Channel.Type.FileTransfer`
`ofdT.Channel.TargetHandleType`	`Handle_Type_Contact` (1)
`ofdT.Channel.TargetID`	`escher@tuxedo.cat`
`ofdT.Channel.Type.FileTransfer.Filename`	`meow.jpg`
`ofdT.Channel.Type.FileTransfer.ContentType`	`image/jpeg`

表 20.3: ファイル転送チャネルのリクエスト

チャネルは、**作成** *(create)* **する**か、あるいは**確保** *(ensure)* **する**ことができる。チャネルを確保するとは、そのチャネルがまだ存在しないときにだけ作成するということである。チャネルを作成しようとした場合は、まったく新しい個別のチャネルを作成する。ただし、すでにそのチャネルが存在する場合にはエラーになる。同じチャネルのコピーは複数存在できないからである。一般的には、テキストチャネルや通話に関しては「確保」を使い (だれかとのやりとりに必要なチャネルはひとつだけだし、実際のところ、たいていのプロトコルは同じ

相手と複数の会話を別々にできるようになっていない)、ファイル転送やステートフルなチャネルなどでは「作成」を使うことになるだろう。

(リクエストなり何なりによって) 新たに作成されたチャネルは、コネクションからのシグナルで通知される。このシグナルには、チャネルの**不変**なプロパティのマップが含まれる。不変なプロパティとは、そのチャネルの活動期間を通して値が変わらないことが保証されているプロパティのことだ。不変であろうとみなされているプロパティについては Telepathy の仕様でそのように明記されているが、一般的にはチャネルの型やターゲット、作成者、インターフェイスなどがそれに含まれる。チャネルの状態などのプロパティは、もちろん不変ではない。

昔ながらのチャネルリクエスト

チャネルのリクエストは元々はシンプルなもので、単に型とハンドルタイプそしてターゲットハンドルを指定するだけのことだった。しかしこれは柔軟性に欠けていた。というのも、すべてのチャネルがターゲットを持っているわけではないし (例: 連絡先検索チャネル)、最初のリクエストのときにそれ以外の情報を必要とするチャネルもあったからだ (例: ファイル転送、ボイスメールのリクエスト、SMS 送信用のチャネル)。

また、チャネルをリクエストするときには二通りの振る舞い (一意なチャネルを新たに作成するのか単に既存のチャネルを確保するだけなのか) を考えないといけないこともわかったし、当時はどっちの振る舞いにするのかを決めるのがコネクションの役割だった。そんなこともあって、かつての方式はとりやめて新しい方式に変わったのだ。こちらのほうがより柔軟で明示的な方式である。

チャネルを作成あるいは確保するときにそのチャネルの不変なプロパティを返すようにしたことで、そのチャネルのプロキシーオブジェクトを高速に作れるようになった。必要な情報をわざわざリクエストする必要がなくなったのだ。表 20.4 は不変なプロパティを示すもので、テキストチャネルをリクエストした (つまり、表 20.3 のチャネルリクエストを使った) 場合の一例だ。TargetHandle や InitiatorHandle など一部のプロパティは、簡潔にまとめるために省略した。

プロパティ	値
ofdT.Channel.ChannelType	Channel.Type.Text
ofdT.Channel.Interfaces	[Channel.Interface.Messages, Channel.Interface.Destroyable, Channel.Interface.ChatState]
ofdT.Channel.TargetHandleType	Handle_Type_Contact (1)
ofdT.Channel.TargetID	escher@tuxedo.cat
ofdT.Channel.InitiatorID	danielle.madeley@collabora.co.uk
ofdT.Channel.Requested	True
ofdT.Channel.Interface.Messages. SupportedContentTypes	[text/html, text/plain]

表 20.4: 新しいチャネルが返す不変なプロパティの例

リクエストする側のプログラムは、あるチャネルへのリクエストを通常は Channel Dispatcher に対して送る。このときに渡すのは、そのリクエストの対象となるアカウントとチャネルリクエストであり、必要なハンドラをオプションで渡すこともある (これは、プログラム側でチャネル自身を扱いたい場合に便利だ)。コネクションではなくアカウントの名前を渡す意味は、Channel Dispatcher から Account Manager に対して、必要に応じてそのアカウントをオンラインにするよう依頼できるようにすることだ。

リクエストが完了すると、Channel Dispatcher がそのチャネルを名前付きのハンドラに渡すか適切なハンドラの場所を示す (ハンドラやその他のクライアントに関しては後述する)。要求するハンドラの名前をオプションにすることで、最初のリクエスト以降の通信チャネルに興味のないプログラムでもチャネルをリクエストできるようになり、最適なプログラムで処理させるようにできる (メールソフトからテキストチャットを立ち上げるなど)。

リクエストする側のプログラムはチャネルリクエストを Channel Dispatcher に送り、Channel Dispatcher はそのリクエストを適切なコネクションに転送する。コネクションは NewChannels シグナルを発行し、これを Channel Dispatcher が受け取ると、そのチャネルを処理できる適切なクライアントを探す。リクエストしたものでない、外部から受信したチャネルのディスパッチも同様に行い、コネクションからのシグナルを Channel Dispatcher が取り上げる。しかし当然、プログラムからの最初のリクエストはない。

図 20.4: チャネルリクエストとディスパッチ

クライアント

クライアントは、送受信チャネルを処理したり監視したりする。クライアントとは、Channel Dispatcher で登録したもののことである。クライアントには次の三種類の形式がある (開発者次第で、あるひとつのクライアントが二種類の形式あるいは三種類すべての形式を兼ねることもある)。

オブザーバー: チャネルとのやりとりをせずにただ観察する。オブザーバーは、チャットに使ったりアクティビティ(VoIP 通話の発信や着信など) の記録に使ったりすることが多い。

アプルーバー: チャネルの受信を許可するか拒否するかをユーザーが決められるようにする。

ハンドラー: 実際にチャネルとのやりとりをする。テキストメッセージを受け入れたり送信したり、ファイルを送受信したりといったものである。ハンドラーは、ユーザーインターフェイスと関連づけられることが多い。

クライアントは、最大で三つまでのインターフェイスを持つ D-Bus サービスを提供する。`Client.Observer`、`Client.Approver` そして `Client.Handler` だ。それぞれのインターフェイスが提供するメソッドを Channel Dispatcher が呼んで、観察あるいは許可あるいは処理させたいチャネルをクライアントに伝える。

Channel Dispatcher は、そのチャネルを各クライアントグループに振り分ける。まず、そのチャネルを適切なオブザーバーすべてにディスパッチする。すべての結果が戻ってきたら、チャネルを適切なアプルーバーすべてにディスパッチする。最初のアプルーバーがチャネルを許可あるいは拒否すると、その他すべてのアプルーバーにそれが通知され、最終的にチャネルはハンドラーに渡される。チャネルのディスパッチがこのように段階を踏んで行われる理由は、オブザーバーの準備をするのにある程度の時間が必要で、それまではハンドラーがチャネルに手を加えられないからである。

クライアントはチャネルフィルタプロパティを公開する。これは Channel Dispatcher が読むフィルタの一覧で、これを読めばそのクライアントがどんなチャネルに興味を持っている

プロパティ	値
ofdT.Channel.ChannelType	Channel.Type.Text
ofdT.Channel.TargetHandleType	Handle_Type_Contact (1)

表 20.5: チャネルフィルタの例

のかがわかる。フィルタに少なくとも含める必要があるのは、チャネル型とそのクライアントが扱えるターゲットハンドル型 (連絡先あるいはルームなど) である。それ以外にもプロパティを含めることができる。このフィルタをチャネルの不変プロパティとマッチングさせる。マッチするかどうかは、単純に一致するかどうかで判断する。たとえば表 20.5 のフィルタは、すべての一対一テキストチャネルにマッチする。

クライアントを D-Bus 経由で発見できるのは、クライアントが既知の名前 `ofdT.Client` から始まる (たとえば `ofdT.Client.Empathy.Chat`) サービスを公開しているからである。また、オプションでファイルをインストールできる。これを Channel Dispatcher が読んで、チャネルフィルタを指定する。これを使えば、Channel Dispatcher からまだ起動していないクライアントを立ち上げることができる。このようにしてクライアントを発見可能にしておくことで、ユーザーインターフェイスを設定や変更がいつでもできるようになり、その時に Telepathy の他の部分に一切手を加える必要がなくなる。

オールオアナッシング

すべてのチャネルを扱えるというフィルタを提供することもできる。しかし、現実的に考えると、そんなことをして有用なのはチャネルを観察するサンプルくらいである。実際のクライアントは、チャネル型に固有のコードを含むものだ。

空のフィルタを渡すと、そのハンドラーは一切のチャネル型に興味を持たないということになる。しかしそれでも、名前を直接指定してそのハンドラに処理をディスパッチできる。特定のチャネルを処理するためだけに一時的に作るハンドラーなどで、このようなフィルタを使うことがある。

20.4　言語バインディングの役割

Telepathy は D-Bus API なので、D-Bus をサポートするあらゆる言語から使うことができる。言語バインディングは Telepathy に必須のものではないが、これを使えば Telepathy をさらに便利に活用できる。

言語バインディングは二つのグループに分類できる。ローレベルのバインディングとハイレベルのバインディングだ。ローレベルのバインディングには、仕様や定数、メソッド名に基づいて生成したコードが含まれる。ハイレベルのバインディングは手書きのコードで、プ

ログラマーが Telepathy をより使いやすくするものだ。ハイレベルのバインディングの例には GLib や Qt4 用のバインディングがある。ローレベルのバインディングの例は Python バインディングや C の libtelepathy であるが、GLib や Qt4 バインディングにもローレベルのバインディングが含まれている。

非同期プログラミング

言語バインディング内でのすべてのメソッド呼び出しは D-Bus 越しのリクエストであり、これは非同期になる。リクエストを投げて、結果はコールバックで取得するという方式だ。D-Bus 自体が非同期なため、これが必須となる。

ネットワークやユーザーインターフェイスがからむプログラミングにはよくあることだが、D-Bus ではイベントループを使う必要がある。入力シグナル用のコールバックへのディスパッチやメソッドの結果を返すときにこれを使う。D-Bus は、GTK+や Qt が使う GLib メインループとうまく統合されている。

D-Bus 言語バインディングの中には、dbus-glib のように疑似同期型 API を提供するものもある。この API では、メインループをブロックしてメソッドの結果を待つ。昔々は、これは telepathy-glib API バインディング経由で公開されていた。残念ながら、疑似同期型の API を使っているといろんな問題に悩まされることになり、最終的に telepathy-glib から削除された。

最初の Telepathy バインディングは C で生成したものだったが、そこでのメソッド呼び出しは単に typedef コールバック関数を使っていた。コールバック関数側では単にそれと同じ型シグネチャを実装しておく必要があるだけだった。

```
typedef void (*tp_conn_get_self_handle_reply) (
    DBusGProxy *proxy,
    guint handle,
    GError *error,
    gpointer userdata
);
```

このアイデアはシンプルだし C でも動くので、次世代のバインディングにも引き継がれた。

最近は、JavaScript や Python のようなスクリプティング言語、そして Vala という C 風の言語を使おうとする人も出てきた。これらは GLib/GObject ベースの API を GObject-Introspection ツール経由で使うものだ。残念ながら、これらのコールバックを他の言語に再バインドするのは極めて難しい。そこで、新たなバインディングを設計することにした。言語自身や GLib の持つ非同期コールバック機能を活用するものだ。

オブジェクトのレディネス

ローレベル Telepathy バインディングのようなシンプルな D-Bus API では、メソッド呼び出しを開始したり D-Bus オブジェクトのシグナルを受信したりするには単にプロキシオブジェ

疑似同期 **D-Bus** 呼び出しが失敗する理由

dbus-glib などの D-Bus バインディングが提供する疑似同期インターフェイスの実装は、request-and-block 方式になっている。ブロックしている間、D-Bus ソケットだけが新たな I/O をポールする。リクエストへの応答以外の D-Bus メッセージはすべてキューに入れておき、後で処理する。

それが原因で、重大かつ不可避な問題がいくつか発生する。

- 呼び出し元はブロックされ、リクエストに対する応答があるまで待たされる。呼び出し元 (そしてそれに対応するユーザーインターフェイス) は、その間は何もできなくなる。ネットワーク越しのリクエストの場合などは、それなりに時間がかかる。呼び出された側がダウンした場合などは、タイムアウトになるまで呼び出し元は身動きできなくなる。

  スレッドを分けたって問題は解決しない。スレッドを分けるということは、単に非同期呼び出しを別の方法で行っているだけだからである。それだったら普通に非同期呼び出しをしたほうがいい。そうすれば、レスポンスは既存のイベントループ経由で得られる。
- メッセージの順番が入れ替わることがあり得る。応答待ちのメッセージの前に受け取ったメッセージはキューに入るので、その応答がクライアントに配送されるのは応答待ちメッセージの処理を終えた後になる。

  これが問題になるのは、状態が変わった (オブジェクトが破棄されたなど) というシグナルを受信する前にそのオブジェクトのメソッド呼び出しを実行してしまう (その結果として `UnknownMethod` が発生する) などの場合だ。こんなときは、ユーザーにどんなエラーメッセージを見せればいいのかわからなくなる。もし事前にシグナルを受け取っていたら、ペンディング中の D-Bus メソッド呼び出しをキャンセルするなりその応答を無視するなりの対策ができる。
- 二つのプロセスがお互いに疑似同期呼び出しをするとデッドロックが発生し、お互いが相手への問い合わせの応答を待ち続ける状態になってしまう。こんな状況が発生するのは、ある D-Bus サービスが別の D-Bus サービス (Telepathy クライアントなど) を呼ぶ場合などである。Channel Dispatcher はクライアントのメソッドを呼んでチャネルをディスパッチする。一方でクライアントも Channel Dispatcher のメソッドを呼んで新たなチャネルのオープンを要求する (あるいは Account Manager を呼ぶこともあって、こちらの場合も同様の可能性がある)。

クトを作るだけでよい。プロキシオブジェクトにオブジェクトへのパスとインターフェイス名を渡すだけで始められる。

しかし Telepathy のハイレベル API の場合は、どんなインターフェイスが使えるかをオブジェクトのプロキシが知っていて欲しいし、オブジェクト型に共通するプロパティ(チャネル型やターゲット、イニシエータなど) も取得できて欲しい。また、オブジェクトの状態や現状(接続状況など) も取得できるようにしたい。

そのために、すべてのプロキシオブジェクトに**レディネス**という概念が存在する。プロキシオブジェクト上のメソッドを呼び出すとそのオブジェクトの状態を非同期で取得できるようになり、状態の取得を終えてオブジェクトが使えるようになった時点で通知される。

指定したオブジェクトの全機能をクライアントが実装しているとは限らないので、オブジェクト型のレディネスはそのオブジェクトで使える機能とは切り離されている。各オブジェクトは**コア**機能を実装している。これは、そのオブジェクトに欠かせない情報 (`Interfaces` プロパティや基本状態) を準備するためのものだ。そしてそれとは別にオプションの機能も実装しており、ここには追加のプロパティや状態の追跡などが含まれる。各種プロキシ上で使える追加機能の例としては、連絡先情報や位置情報、チャットのステータス (“Escher is typing…” など) そしてユーザーのアバターなどがある。

たとえば、コネクションオブジェクトのプロキシが持つ機能は、次のようになる。

- インターフェイスや接続状態を取得するコア機能。
- リクエスト可能なチャネルのクラスやサポートする連絡先情報などを取得する機能。
- 接続を確立して、接続完了通知を返す機能。

プログラマーは、そのオブジェクトを使う準備が完了したかどうかを問い合わせる。その際に、使いたい機能のリストと準備完了時に呼び出すコールバックを指定する。すべての機能が既に使える状態の場合はコールバックが即時に呼び出されるが、そうでない場合は、指定したすべての機能の準備が整った時点でコールバック呼び出される。

20.5　頑健性

Telepathy の大きな利点のひとつが、その頑健性だ。コンポーネントがモジュール化されているので、なにかひとつのコンポーネントがクラッシュしたとしてもシステム全体がダウンするようなことがない。Telepathy の頑健性の理由を以下にまとめる。

- Account Manager と Channel Dispatcher が自身の状態を復元できること。Mission Control(Account Manager と Channel Dispatcher を含む単一のプロセス) が立ち上がるときに、ユーザーのセッションバスに現在登録されているサービス名を調べる。既知のアカウントに関連づけられているコネクションが見つかれば (新たな接続を確立せずに) そのアカウントとの関連づけをもう一度行い、稼働中のクライアントには自分が扱っているチャネルのリストを問い合わせる。

- 処理中のチャネルがオープンしているときにクライアントが消えてしまった場合は、Channel Dispatcher がクライアントをもう一度立ち上げてチャネルを再発行する。

  クライアントが頻繁にクラッシュする場合は、もし別のクライアントがあるのなら Channel Dispatcher は別のクライアントを立ち上げることもできる。別のクライアントがない場合は、チャネルを閉じる (そして、扱うことのできないデータを渡されてクライアントがクラッシュしまくるのを防ぐ)。

  テキストメッセージの場合、受信確認があってからでないと処理待ちメッセージのリストからは消せない。クライアント側でメッセージの受信を確認できるのは、ユーザーがメッセージを見たとき (つまり、フォーカスのあたっているウィンドウにメッセージが表示されたとき) だけである。この方式にしておくと、もしメッセージのレンダリング中にクライアントがクラッシュしたとしてもそのメッセージはまだ表示されていないものとして処理待ちキューに残る。
- コネクションがクラッシュした場合は Account Manager がコネクションを立ち上げなおす。ステートフルなチャネルの内容は当然消えてしまうが、そのプロセスで稼働していたコネクションにしか影響は及ばず、他には影響しない。クライアントはコネクションの状態を監視できるので、連絡先リストやステートレスなチャネルなどに関しては単純に情報を再リクエストするだけでよい。

20.6 Telepathy の拡張: サイドカー

Telepathy の仕様では、各種通信プロトコルが公開するさまざまな機能をできるだけカバーしようとしている。しかし中には、プロトコル自体が拡張可能なものもある [4]。Telepathy の開発者たちは、Telepathy コネクションを拡張してそれに対応できるようにしようとした。Telepathy の仕様自体はそのままにしておきたかったのだ。これを実現するのが**サイドカー**である。

サイドカーは、一般に Connection Manager のプラグインとして実装されている。クライアントからは、指定した D-Bus インターフェイスを実装したサイドカーをリクエストするメソッドを呼ぶ。たとえば誰かが XEP-0016 プライバシーリストを実装したとすると、そのインターフェイスは `com.example.PrivacyLists` のようになるだろう。このメソッドは、プラグインが提供する D-Bus メソッドを返す。このオブジェクトは指定したインターフェイスを実装していなければ行けない (それ以外のインターフェイスも実装しているかもしれない)。このオブジェクトは、メインのコネクションオブジェクトと並んで存在する (だからこそサイドカーという名前にした。ちょうどバイクのサイドカーと同じようなものだ)。

[4] Extensible Messaging and Presence Protocol (XMPP) がその一例だ。

サイドカーの歴史

Telepathy が誕生して間もないころ、One Laptop Per Child プロジェクトがカスタム XMPP エクステンション (XEP) を必要としていた。デバイス間で情報を共有するためのものだ。そのエクステンションは Telepathy-Gabble (XMPP Connection Manager) に直接追加され、コネクションオブジェクトの非公式なインターフェイスとして公開された。後に、他の開発者たちも別の XEP への対応を求めるようになった。類似機能が他の通信プロトコルにはないようなものだ。そんな経緯があって、プラグイン用のより汎用的なインターフェイスが必要だという結論に達したのだ。

20.7 コネクションマネージャーの内部構造の概要

ほとんどの Connection Manager は C/GLib バインディングを使って書かれており、高レベルの基底クラス群の多くが Connection Manager を書きやすくするために開発されたものだ。先に説明したとおり、D-Bus オブジェクトを公開するのはさまざまなインターフェイスを実装したソフトウェアオブジェクトであり、それらのインターフェイスを D-Bus インターフェイスとマップしている。Telepathy-GLib は、Connection Manager やコネクションそしてチャネルを実装するための基底オブジェクトを提供する。また、Channel Manager を実装するためのインターフェイスも用意する。Channel Manager はファクトリーで、`BaseConnection` によってインスタンス化され、バス上に公開するチャネルオブジェクトを管理する。

C/GLib バインディングは、いわゆる *mixin* も提供する。これをクラスに追加すると、付加機能を提供できる。そして API の仕様を抽象化し、新たな API が追加されたり過去の API が廃止されたりしても互換性を保てるようになる。最も使われている mixin は、D-Bus プロパティのインターフェイスをオブジェクトに追加するものだ。それ以外にも、`ofdT.Connection.Interface.Contacts` や `ofdT.Channel.Interface.Group` インターフェイスを実装する mixin もあれば、新旧のプレゼンスインターフェイスや新旧のテキストメッセージインターフェイスを同じメソッド群で実装できるようにする mixin もある。

20.8 教訓

Telepathy は、モジュール化された柔軟な API を D-Bus 上に実装したすばらしい実例のひとつだ。拡張性があって疎結合なフレームワークを D-Bus 上で開発する方法を示している。中央集権型のデーモンを使わずにコンポーネントを再起動できるし、他のコンポーネントのデータを失うこともない。Telepathy はまた、D-Bus を効率的に使う方法も示している。バス上をやりとりするデータのトラフィックを最小限に抑える方法がわかるだろう。

図 20.5: コネクションマネージャーのアーキテクチャの例

Mixin による、API の間違いの解決

mixin を使って Telepathy の仕様バグを解決した例が TpPresenceMixin だ。元々Telepathy が公開していたインターフェイス (odfT.Connection.Interface.Presence) は信じられないほど複雑で、コネクションとクライアントのどちらを実装するのも大変だった。また、そもそもほとんどのプロトコルでコネクションにもクライアントにも存在しない機能を公開していたり、片方では使うけれどももう一方ではほとんど使わないという機能を公開していたりした。後にこのインターフェイスはもっとシンプルなもの (odfT.Connection.Interface.SimplePresence) に置き換えられた。これはユーザーが必要としていて Connection Manager で実際に実装されている機能はすべて公開するものだ。

プレゼンス mixin はコネクション上でのどちらのインターフェイスも実装しているので、レガシーなクライアントでもドウサスル。しかし、機能面ではシンプルなインターフェイスのレベルにとどまる。

Telepathy の開発は漸進的に行われており、時を経て D-Bus の使い方も改善されている。かつては間違いも犯したし、そこから得た教訓もあった。Telepathy のアーキテクチャ設計で学んだ大切なことを、ここでいくつか紹介する。

D-Bus のプロパティを使うこと。細かい D-Bus メソッド呼び出しを大量に行わないと情報が得られないというのはやめる。あらゆるメソッド呼び出しにはラウンドトリップタイム

が発生する。個別の呼び出し (GetHandle、GetChannelType、GetInterfaces など) を大量に行うのではなく、D-Bus プロパティを使ってすべての情報を一回の GetAll だけで返せるようにする。

新たなオブジェクトをアナウンスするときには可能な限りの情報を提供する。新たなオブジェクトを知ったクライアントが最初にするのは、そのすべてのプロパティをリクエストしてそのオブジェクトで何ができるのかを知ることだ。オブジェクトを通知するシグナルの中に不変なプロパティを含めておけば、たいていのクライアントはそれだけでどんなことができるのかを判断できる。さらに、そのオブジェクトを実際に使うことになっても、不変なプロパティを改めてリクエストする手間が省ける。

Contacts インターフェイスは、複数のインターフェイスからのリクエストを一度に受け付けられるようにする。大量の GetAll 呼び出しで連絡先の全情報を取得するのではなく、Contacts インターフェイスがすべての情報を一度に取得できるようにしておけば D-Bus のラウンドトリップを減らせる。

すべきでないところでの抽象化はやめる。Group インターフェイスを実装したチャネルで連絡先一覧や連絡先グループを公開するというのはよい考えに思える。新たなインターフェイスを追加せずに既存の抽象化を使えるからだ。しかし、クライアント側の実装は難しくなってしまうので、結局はあまり適しているとは言えない。

将来のニーズを満たせるように API を設計する当初のチャネルリクエスト API は非常に柔軟性に欠け、ほんとうに基本的なチャネルリクエストしかできなかった。もっと多くの情報を必要とするチャネルリクエストをしたいというニーズは満たせないものだった。この API は書き直さざるを得なくなり、ずっと柔軟なものに置き換えられた。

第21章

Thousand Parsec

Alan Laudicina and Aaron Mavrinac

広大なスターエンパイアには100の世界が存在し、1000パーセクにまたがって広がっている。銀河の他の部分とは違って、ここには戦士はほとんどいない。そこにいるのは知的な人々で、豊かな文化と教養を兼ね備えている。彼らの住む惑星では科学や工学の大学が続々とうまれ、この世代の平和と繁栄の象徴となっている。象限を超えて遙か彼方から宇宙船がやってきて、一線級の研究者たちが隅々まで調査している。彼らはその知識をいかして、これまでにない壮大なプロジェクトにとりかかる。分散型のコンピューターネットワークを構築して銀河全体をひとつにするというプロジェクトだ。さまざまな言語や文化、そして法体系をすべてひとつにつないでしまおうというのだ。

Thousand Parsecは単なるビデオゲームではない。一種のフレームワークであり、マルチプレイヤー型でターン制の宇宙帝国戦略ゲームを構築するための完全なツールキットを含む。汎用的なゲームプロトコルを使えばさまざまな実装のクライアントやサーバーそして AI ソフトウェアを作れるし、あらゆるゲームを作れるだろう。規模が規模だけに実際に計画を実行するのは大変だろうが、オープンソースアプリケーションのアーキテクチャについて議論するには興味深いものになる。

Thousand Parsecのようなゲームは、ジャーナリストに言わせると“4X” (“e*x*plore, e*x*pand, e*x*ploit, and e*x*terminate” の略) というジャンルらしい。プレイヤーが帝国をコントロールするものだ[1]。一般的な4X系のゲームでは、プレイヤーたちが地図を探し (explore)、新たな協定を結んだり既存の協定の効力を拡張したり (expand)、自分たちでコントロールできる資源を活用したり (exploit) して、ライバルのプレイヤーたちを攻撃して滅亡させる (exterminate)。経済成長や技術開発の重視、マイクロマネージメント、そしていろんなルートの優位性などによって、このジャンルで他に類を見ない深みと複雑さを生み出した。

プレイヤーの視点でとらえると、三つの主要なコンポーネントがThousand Parsecのゲー

[1] Thousand Parsecの発想の元になった、すばらしい商用ゲームの例がある。*VGA Planets* や *Stars!*、そして *Master of Orion* や *Galactic Civilizations* や *Space Empires* シリーズなどである。どれもあまりなじみのない人がいるかもしれない。では *Civilization* シリーズならどうだろう。これも同じスタイルのゲームだが、設定が少し異なる。リアルタイム型の4Xゲームもいろいろあって、たとえば *Imperium Galactica* や *Sins of a Solar Empire* がその一例だ。

ムにかかわることになる。まず最初がクライアント。プレイヤーがゲームの世界とのやりとりをするアプリケーションだ。クライアントはネットワーク越しに (あらゆる重要なプロトコルを用いて) サーバーに接続する。他のプレイヤー (場合によっては人工知能) のクライアントもそこに接続している。そしてサーバー。サーバーでは全体的なゲームの状態を管理し、各ターンの開始時にクライアントを更新する。プレイヤーはさまざまなアクションを行い、それをサーバーに返す。サーバーでは、その結果を計算して次のターンに備える。そして最後がルールセット。プレイヤーが行うアクションの性質はルールセットによって決まる。突き詰めると、このルールセットがゲームを定義することになる。これはサーバー側で実装するもので、それをサポートするクライアントによってプレイヤー側で実現する。

さまざまなゲームが作られるだろうし、その多様性に対応するだけの複雑なアーキテクチャが必要となる。Thousand Parsec は、ゲーマーだけでなく開発者にとってもエキサイティングなプロジェクトだ。ゲームのフレームワークの仕組みになんか興味がないというまじめな開発者さんたちにも、ぜひそのおもしろさを知っていただきたい。その裏側で動いているクライアント・サーバー通信や動的な構成、メタデータの処理、レイヤー化した実装など、これらすべてが長年をかけて有機的に成長してきた。典型的なオープンソーススタイルでだ。

本質的には、そもそも Thousand Parsec はゲームのプロトコルやその他関連機能に関する標準仕様群である。本章の大半は、この視点でとらえたフレームワークについて議論する。しかし、実際の実装を参照したほうがずっとわかりやすい場合も多い。そのために私たちは、主要コンポーネントのそれぞれについて「フラッグシップ」実装を選び、それを使って具体的な議論をすることにした。

モデルとなるクライアントは `tpclient-pywx` だ。これは比較的成熟した wxPython ベースのクライアントで、現時点では最も多くの機能をサポートしており、最新バージョンのゲームプロトコルにも対応している。このクライアントは `libtpclient-py` によってサポートされている。これは Python のクライアントヘルパーライブラリで、キャッシュなどの機能を提供する。また、Python のライブラリ `libtpproto-py` は最新バージョンの Thousand Parsec プロトコルを実装している。サーバーには `tpserver-cpp`。これも歴史のある C++の実装で、最新の機能やプロトコルに対応している。これを見本として採用する。このサーバーはさまざまなルールセットに対応しているが、中でもマイルストーンルールセット *Missile and Torpedo Wars* は、「伝統的な」4X 宇宙ゲームを実装するためのよい見本となる。

21.1　スターエンパイアの世界

Thousand Parsec の世界を構成するものごとを適切に紹介するために、まずこのゲームの概要を説明しよう。ここでは *Missile and Torpedo Wars* ルールセットを見ていく。このルールセットはプロジェクトの二番目のマイルストーンルールセットで、現在の安定版 Thousand Parsec プロトコルにある主要な機能のほとんどを使っている。あまりなじみのない用語もいくつか登場するが、それらは追々解説していくので理解してもらえるだろう。

Missile and Torpedo Wars は上級者向けのルールセットで、Thousand Parsec フレームワークで使えるすべてのメソッドを実装している。本章の執筆時点ではそんなルールセットはこれだけしかない。また、手早く拡張し、より完全な楽しいゲームになった。

Thousand Parsec サーバーとの接続を確立するときに、クライアントはゲームエンティティの一覧を調べて全カタログをダウンロードする。このカタログにはすべてのオブジェクトやボード、メッセージ、カテゴリ、デザイン、コンポーネント、プロパティ、プレイヤー、そしてゲームの状態を形作る資源が含まれている。これらの詳細については後ほど詳述する。ゲームを始めるときにクライアント側で知るべきことが大量にあるように見える (各ターンの終了時も同様だ) が、これらの情報はどれもゲームに不可欠なものである。この情報をダウンロードし終えるには数秒程度かかるが、これでクライアントに必要な情報はすべてそろい、ゲームの世界を表現できるようになる。

最初にサーバーに接続したときにランダムに惑星が作られ、それが新しいプレイヤーの「母星」となる。そして、その母星には自動的にふたつの艦隊が配備される。各艦隊はふたつの Scout デザイン (デフォルトのデザイン) で構成されており、Scout Hull と Alpha Missile Tube を持っている。Explosive コンポーネントが追加されていないので、デフォルトの艦隊はこのままでは対艦隊戦や対惑星戦ができない。要するに、まったく無防備な状態だ。

この時点でプレイヤーがすべきことは、まず艦隊の武装を始めることだ。Build Weapon 命令を使って兵器を設計して、できあがった製品を Load Armament 命令で艦隊に搭載する。Build Weapon 命令は、惑星の資源—各惑星には、資源がランダムに割り当てられている— を製品に変換する。爆発性の弾頭で、これを惑星の地表で生産するわけだ。Load Armament 命令は、完成した製品を艦隊に転送する。

惑星の地表にある資源を使い切ってしまったら、新たな資源を得るための採掘が重要になる。資源には、地表にある状態以外に二種類の状態がある。採掘可能 (mineable) とアクセス不能 (inaccessible) だ。Mine 命令を惑星上で実行すると、採掘可能な資源を地表の資源に変換する。これにはある程度の時間がかかり、地表の資源になれば実際に使えるようになる。

オブジェクト

Thousand Parsec の世界では、形を持つものはすべてオブジェクトとなる。実際、その世界自体もまたオブジェクトだ。このように設計したおかげで、事実上無限の要素をゲームに含めることができるようになり、いくらオブジェクトを増やしたところでその一部しか使わないルールセットはシンプルなままに保てるようになった。新たなオブジェクト型を追加すると、それに固有の情報を格納できるようになる。そしてそれを、Thousand Parsec プロトコルで送信したり使ったりできるというわけだ。現在デフォルトで提供されている基本的なオブジェクト型は Universe、Galaxy、Star System、Planet、そして Fleet の五つである。

Universe は Thousand Parsec のゲームの最上位にあるオブジェクトであり、全プレイヤーが常にアクセスできる。Universe オブジェクト自体は実際のところゲーム全体を取り仕切るわけではなく、非常に重要なたったひとつの情報、つまり現在のターン番号を格納している。

Thousand Parsec の世界ではターン番号のことを “year” と呼んでおり、ターンが終わるごとにひとつずつ増えていく。符号なし 32 ビット整数で格納されるので、4,294,967,295 年までゲームを続けることができる。ただ、理屈上ではそうだが、作者は実際にゲームがそこまで進むのを見たことはない。

Galaxy は、近くにあるオブジェクト (Star System や Planet、Fleet) をとりまとめるコンテナで、それ以外に特別な情報を提供するわけではない。一つのゲームの中には多数の Galaxy が存在することがあり、それぞれが Universe の一部を構成する。

これら二つのオブジェクトと同様に、Star System もその主目的は下位レベルのオブジェクトのコンテナとなることだ。ただ、Star System オブジェクトはクライアントに表示されるオブジェクトの最初の層となる。このオブジェクトの中には、(少なくとも一時的には)Planet や Fleet が含まれることになる。

Planet は大きな天体で、人が住んだり資源を採掘したり設備を生産したり地上軍を配備したりといったことができる。Planet は、プレイヤーが所有できるオブジェクトの最初の層である。ある Planet を所有するというのは気軽にできることではない。たいていのルールセットでは、ひとつも Plante を持っていないということはそのプレイヤーが敗北したことを意味する。Planet オブジェクトには比較的大量のデータが格納される。次のようなものだ。

- その Planet の所有者のプレイヤー ID(所有者がいない場合は-1)。
- その Planet の資源の一覧。資源 ID(型) と、その Planet における地表資源・採掘可能資源・アクセス不能資源の量。

これまでに取り上げた組み込みのオブジェクト群を使えば、伝統的な 4X 型宇宙ゲームに沿った多くのルールセットにとっての十分な基盤となる。当然ながら、ソフトウェア工学の原則に従って、これらのオブジェクトのクラスをルールセット内で拡張することもできる。つまりルールセットを作る人は、新たなオブジェクト型を作ることもできるし既存のオブジェクト型に追加情報を持たせることもできるというわけだ。ゲーム内に登場するあらゆるオブジェクトについて、事実上無制限に拡張できることになる。

命令

命令はルールセットが定義するもので、Fleet オブジェクトと Planet オブジェクトにアタッチできる。コアサーバーにはデフォルトの命令型は何も含まれていないが、どんなに基本的なゲームであっても命令は不可欠なものだ。そのルールセットの性質に沿って、命令を使えばほぼあらゆるタスクを達成できる。4X 型のゲームではお決まりの命令がいくつかあって、それは大抵のルールセットに実装されている。Move、Intercept、Build、Colonize、Mine そして Attack といった命令である。

4X のうちの最初の任務 (つまり explore) をこなすには、ゲーム世界のマップ上を移動できないといけない。これは一般に Move 命令の仕事で、この命令は Fleet オブジェクトに追加されている。Thousand Parsec フレームワークは柔軟で拡張性を重視しているので、Move 命令

の実装もルールセットごとに変更できる。*Minisec* や *Missile and Torpedo Wars* の場合、Move 命令は三次元空間上の座標をパラメータとして受け取る。サーバー側では移動所要時間を計算し、必要なターン数をクライアントに返す。ルールセットによっては、Move 命令が疑似 Attack 命令として機能するものもある。チーム作業が実装されていないようなルールセットでよくあることだ。たとえば Minisec や Missile and Torpedo Wars の場合、敵の艦隊が抑えている座標に移動するとそこで戦いが始まる。ルールセットによっては、Move 命令のパラメータが違っている (つまり、三次元座標を使わない) ものもある。たとえば *Risk* ルールセットには 1 ターンでの移動しか存在しない。「ワームホール」でつながった別の惑星に直接移動するというものだ。

一般的に Fleet オブジェクトに追加されることが多い Intercept 命令は、空間内の他のオブジェクト (たいていは敵艦隊) に遭遇させるために使うものだ。Move 命令と似ているが、実行している間にふたつのオブジェクトが別方向に移動する可能性もあるので、単純に空間内の座標を指定するだけでは他の艦隊にぶつけるのは不可能だ。そこで、別の命令を用意する必要があった。この問題に対応するために用意されたのが Intercept 命令で、これを使えば深宇宙にいる敵艦隊を一掃したり、危機的な状況で攻撃されるのをかわしたりできる。

Build 命令は、4X のうちの二つ (e*x*pand と e*x*ploit) に対応するものだ。ゲーム世界で勢力を拡大していく手段として明白なのは、大量の艦隊を編成してより遠方まで進出することだ。Build 命令は Planet オブジェクトに追加されるのが一般的で、たいていは、その惑星に存在する資源の量 (そして採掘状況) による制約を受ける。幸運にも資源に恵まれた母星を持っているプレイヤーは、ゲームの初期段階で有利にたてる。

Build 命令と同様に、Colonize 命令も e*x*pand と e*x*ploit を実行するためのものだ。ほとんどの場合は Fleet オブジェクトに追加される命令であり、この Colonize 命令を使うと、まだ誰も領有権を宣言していない惑星を占領できる。ゲーム世界の中で、自分の制御のおよぶ範囲を拡大するのに役立つだろう。

Mine 命令は e*x*ploit のために使うものだ。この命令は Planet オブジェクトあるいはその他の天体オブジェクトに追加されるのが一般的で、まだ地表にはない資源を採掘することができる。採掘を実行すれば資源が地表に登場し、後でそれを使って開発をしたり、ゲーム世界内でプレイヤーの手の届く範囲を広げたりする。

いくつかのルールセットに実装されている Attack 命令を使うと、敵の Fleet や Planet との戦闘を明示的に始めることができる。これは、4X の最後の任務すなわち e*x*terminate を達成するためのものだ。チームベースのルールセットでは、(Move や Intercept で攻撃対象を指定して暗黙のうちに戦いを始めるのではなく) Attack 命令を別途用意しておくことが重要となる。そうしないと、まちがって友軍との戦闘が始まってしまうことがあり得るからである。

Thousand Parsec フレームワークでは、ルールセットの開発者が自分で命令の型を定義できるようにしているので、どこにもないような斬新な命令を作ることもできる…というか、それが推奨されている。追加のデータをあらゆるオブジェクトに組み込めるので、開発者にとっては新たな命令を作るのはとてもおもしろいことだろう。

資源

資源は、ゲーム内のオブジェクトに組み込まれているデータのひとつである。特に Planet オブジェクトで幅広く使われており、これを使うとルールセットを簡単に拡張できる。Thousand Parsec の設計の多くがそうであったように、資源という概念を含めたのも拡張性を考慮したためだ。

資源を実装するのはルールセットを設計する人であることが多いが、フレームワーク全体を通して一貫して使われる資源がひとつだけある。それが Home Planet で、これはプレイヤーの母星を識別するために使われる。

Thousand Parsec のベストプラクティスに従うと、資源の一般的な利用法は、何らかの型のオブジェクトに変換できる何かを表現することだ。たとえば Minisec では Ship Parts という資源を実装しており、ゲーム世界のそれぞれの Planet オブジェクトに対してランダムな量が割り当てられる。どこかの惑星を占領すると、その惑星の Ship Parts を使って Build 命令で Fleet に変換することができる。

Missile and Torpedo Wars は、おそらく現時点で最も資源を活用しているルールセットだろう。兵器を動的な資源として扱った最初のルールセットでもある。これはつまり、惑星上にある兵器を船に追加したり、逆に船に搭載していた兵器を惑星に戻したりできるということだ。ゲーム内のそれぞれの兵器に対して新たな資源の型を用意し、ゲーム内で資源を作るようにしたのだ。これで、資源の型を指定して船上の兵器を特定したり、ゲームの世界の中で兵器を自由に運べるようになった。*Missile and Torpedo Wars* では、工場 (惑星上での生産施設) についても Factories という資源で管理しており、これを各惑星に結び付けている。

デザイン

Thousand Parsec では、兵器や船はさまざまなコンポーネントで構成されている。こういったコンポーネントを組み合わせたものが、デザインの基盤となる。デザインとは、ゲームの中で組み立てたり使ったりする何かのプロトタイプのことだ。ルールセットを作ろうとすると、デザイナーはこんな決断を迫られる。そのルールセットで兵器や船のデザインを動的に作れるようにする？それとも、あらかじめ用意したデザインしか使えないようにする？用意されたデザインしか使えないようにしてしまえばルールセットを作るのも簡単になるしバランスも保ちやすい。一方、動的にデザインを作れるようにすれば、ルールセットを作るのは複雑で難しくなるが、そのぶんゲームは楽しくなる。

ユーザーがデザインを作れるようにすると、ゲームはより高度なものになる。ユーザーは自分の船やその武装を戦略的に設計しないといけないので、ゲームにいっそうの変化が加わることになる。プレイヤーが配置された場所がたまたまよい場所だったとか、その他のゲーム戦略上得られた利点などを打ち消す助けになるだろう。これらのデザインは各コンポーネントのルールの影響を受ける。コンポーネントのルールは後述する Thousand Parsec Component Language (TPCL) で記述されており、各ルールセットに固有のものである。要するに、兵器

や船のデザインを実装するときに、新たに機能をプログラミングする必要がないということだ。ルールセットで定義されているコンポーネント用にシンプルなルールを設定するだけでよいことになる。

念入りに計画してきちんとバランスを取らないと、カスタムデザインを使う利点も台無しになってしまう。ゲームが後半に進むと、多くの時間を割いて新型の兵器や船のデザインをすることになる。クライアント側でのデザインの操作をうまくできるようにするもまた大変な作業だ。デザインの操作はゲームの中でも重要な位置を占めるが、その他の部分とはあまり関係がない。デザインウィンドウをクライアントに統合してしまうと相当邪魔になってしまう。Thousand Parsec で最も完成度の高いクライアントである tpclient-pywx の場合は、デザインウィンドウのランチャーを比較的目立たない場所に用意している。メニューバーのサブメニューの中で、ゲーム中にはめったに使わないところだ。

デザイン機能は、ルールセットの開発者が簡単に使えるように作られている。これを使えば、ゲームを拡張して事実上無制限に複雑化できる。既存のルールセットの多くは、事前に用意されたデザインしか使えないようになっているが、*Missile and Torpedo Wars* は兵器や船をさまざまなコンポーネントでデザインできるようになっている。

21.2　Thousand Parsec プロトコル

Thousand Parsec プロトコルは、プロジェクトを組み立てるのに必要なあらゆるものの基盤であるという人もいる。このプロトコルでは、ルールセットの作者が使える機能やサーバー側の動き、そしてクライアント側でなにを処理できるのかなどを定義する。最も大切なのは、惑星間通信の標準規格のように、各種ソフトウェアコンポーネントがお互いに理解し合えるようにすることだ。

サーバーでは、ゲームの動的な状態をルールセットの指示に従って管理する。ターンごとに、プレイヤーのクライアントはゲームの状態に関する何らかの情報を受け取る。オブジェクトとその所有者そして現状、進行中の命令、資源の貯蔵量、技術開発の進捗、メッセージ、その他プレイヤーが見るあらゆるものに関する情報だ。プレイヤーは、現状で与えられた選択肢の中から何らかのアクションを実行する。命令を実行したりデザインを作ったりといったものだ。そのアクションがサーバーに返され、サーバー側では次のターンに向けた計算処理を行う。これらすべてのやりとりが、Thousand Parsec プロトコルの枠組みの中で行われるのだ。このアーキテクチャのおかげで、興味深い効果が得られる。AI(人工知能) クライアント—サーバーやルールセットとは別に、コンピューター側のプレイヤーとしてゲームに参加するだけのクライアント— も人間のプレイヤーと同様のルールに縛られるので、人間が見ることのできない情報にこっそりアクセスしたりルールをねじ曲げたりといったいわゆる「チート」ができなくなるのだ。

プロトコルの仕様では、一連のフレームを定義している。これは階層型になっており、Header フレームを除く各フレームが基底フレーム型を持っている。基底フレーム型に対して独自の

データを追加する方式だ。さまざまな抽象フレーム型も用意されている。これらは明示的に使うものではなく、単に具象フレームを作るときのベースとして使うためだけのものだ。フレームには方向の指定も含まれている。これは、ひとつのフレームが、サーバーあるいはクライアントのどちらか一方からもう一方への送信にだけ対応すればよいというようにするためのものである。

Thousand Parsec プロトコルは TCP/IP 越しでそれ単体でも機能するし、HTTP のような別のプロトコル上のトンネル経由でも機能するように作られている。また、SSL による暗号化にも対応している。

基本

いくつかの汎用的なフレームをプロトコルで提供している。これは、クライアントとサーバーとの間の通信で常に利用するものだ。先述の `Header` フレームは、単に他のすべてのフレームの基盤となるためだけのもので、これを直接継承したフレーム `Request` と `Response` 経由で基盤を提供する。`Request` フレームは (クライアントからサーバー、あるいはサーバーからクライアントへの) 通信を開始するときの基盤となり、`Response` フレームはその通信が引き起こすものに対応するフレームとなる。`OK` フレームと `Fail` フレーム (どちらも `Response`) は、通信の中での Boolean ロジックの二つの値を表す。`Sequence` フレーム (これも `Response`) は、リクエストに対するレスポンスが複数のフレームになることを受信者に通知する。

Thousand Parsec は、何かを表すときに数値の ID を利用する。そのため、ID を指定してデータのやりとりをするフレームも用意されている。`Get With ID` フレームは、ID を使って作業をするための基本的なリクエストとなる。それ以外にも `Get With ID and Slot` フレームもある。これは、ID を持つ親の「スロット」内にあるもの (あるオブジェクトに対する命令など) を扱うリクエストだ。もちろん、ID のシーケンスを取得する必要に迫られることもある。最初にクライアントの状態を設定するときなどである。こんなときには、`Get ID Sequence` 型のリクエストと `ID Sequence` 型のレスポンスを利用する。複数のアイテムをやりとりするときによく使われるのが `Get ID Sequence` リクエストと `ID Sequence` レスポンスで、それに続けて一連の `Get With ID` リクエストを実行して個々のアイテムを表すレスポンスを得る。

プレイヤーとゲーム

クライアントがゲームに参加するには、いくつかの手続きが必要だ。まず最初に、サーバーに `Connect` フレームを発行しなければいけない。これに対してサーバーは `OK` あるいは `Fail` で応答する。`Fail` になるのは、たとえば `Connect` フレームに含まれるクライアント側のプロトコルのバージョンがサーバー側と一致しない場合などである。サーバー側では `Redirect` フレームを返すこともできる。他のサーバーに移動させたりサーバープールを使ったりする場合だ。次にクライアントが発行する必要があるのが `Login` フレームだ。ここでプレイヤー

の認証情報を指定する。はじめてそのサーバーに接続するプレイヤーの場合は、サーバー側でアカウントの作成を許している場合はまず Create Account フレームを使える。

Thousand Parsec は常に変化し続けるので、プロトコルにある機能がサーバー側でサポートされているかどうかを何らかの手段で確かめる必要がある。そのために使うのが Get Features リクエストと Features レスポンスだ。サーバー側が返すレスポンスには次のような情報が含まれる。

- SSL および HTTP のトンネリングが (このポートあるいは別のポートで) 使えるかどうか。
- サーバー側でのコンポーネントのプロパティの算出をサポートしているか。
- レスポンスにおける ID シーケンスの並び順 (昇順あるいは降順)。

同様に、Get Games リクエストとそれに対応する一連の Game レスポンスが、クライアントに対してそのサーバー上でアクティブなゲームの特性を示す。一つの Game フレームには、あるゲームに対する次のような情報が含まれる。

- そのゲームの長い (説明的な) 名前。
- サポートするプロトコルのバージョンの一覧。
- サーバーの型とバージョン。
- ルールセットの名前とバージョン。
- 利用可能なネットワーク接続設定の一覧。
- オプションの項目 (プレイヤー数やオブジェクト数、管理者の詳細情報、コメント、現在のターン番号など)。
- ベームが使うメディア用のベース URL。

もちろん、プレイヤーにとっても、自分の対戦相手 (あるいは仲間) の情報を知ることが重要だ。そのためのフレームも用意されている。一般的な項目パターンの交換をするには、まず Get Player IDs リクエストに対して List of Player IDs レスポンスを得て、それから各プレイヤーについての Get Player Data リクエストと Player Data レスポンスを利用する。Player Data フレームには、プレイヤーの名前と種族が含まれている。

ゲームでのターンもこのプロトコルによって制御される。プレイヤーが何らかのアクションを終えると、次のターンの準備が整ったことを表す Finished Turn リクエストを送る。全プレイヤーがこのリクエストを送り終えた時点で次のターンに向けた計算が始まる。ターンには、サーバーで定めた時間制限もある。そのため、反応が遅れたり応答がなかったりしたプレイヤーはゲームについて行けなくなる。クライアントは一般に、Get Time Remaining リクエストを発行して、ローカルのタイマーの値をサーバーの Time Remaining レスポンスに合わせる。

最後に、Thousand Parsec ではさまざまな目的で使うためのメッセージに対応している。ゲームから全プレイヤーへの通知、ゲームから特定のプレイヤーへの通知、そしてプレイヤーどうしのコミュニケーションなどに使うものだ。これらは “board” コンテナが取り仕切ってお

り、ここでメッセージの順序や表示を管理している。項目の並びのパターンに従って、Get Board IDs リクエストと List of Board IDs レスポンス、そして一連の Get Board リクエストと Board レスポンスが行われる。

クライアントがメッセージボードの情報を得たら、Get Message リクエストを使ってボード上のメッセージをスロットで取得できる (従って、Get Message は基底フレームとして Get With ID and Slot を使う)。サーバーは Message フレームを返す。この中にはメッセージのタイトルと本文やそのメッセージが作られたターン、メッセージ内で言及されているエンティティへの参照などが含まれる。Thousand Parsec で通常現れるアイテム (プレイヤーやオブジェクトなど) に加えて、特殊な参照が存在する。メッセージの優先度やプレイヤーのアクション、その他のステータスなどである。もちろん、クライアントは Post Message フレーム (Messsage フレームを運ぶメディア) でメッセージを追加できる。メッセージを削除するときに使うのは Remove Message フレーム (基底フレームは GetMessage) だ。

オブジェクト、命令、そして資源

ゲーム世界でのやりとりの多くは、オブジェクトや命令そして資源に対する各種機能を含むフレームを使って行う。

クライアントは、接続時そして各ターンの終了時に世界の物理的な状態—少なくとも、そのプレイヤーから見えて制御できる範囲の状態—を取得する必要がある。クライアントは一般に、Get Object IDs リクエスト (Get ID Sequence) を発行し、サーバーはそれに対して List of Object IDs で応える。クライアントが個々のオブジェクトの詳細を知るには Get Object by ID リクエストを使う。その応答は Object フレームになり、そのプレイヤーから見える範囲の詳細が含まれている。その型や名前、サイズ、位置、速度、内包するオブジェクト、使える命令、現在の命令などである。プロトコルでは Get Object IDs by Position リクエストも用意している。これを使うと、指定した球体の内部にある全オブジェクトを探すことができる。

クライアントが使える命令を取得するには、通常のアイテム群のときと同様のパターンを使う。つまり、まず Get Order Description IDs リクエストを発行し、返ってきたレスポンス List of Order Description IDs の中の各 ID に対して Get Order Description リクエストをして Order Description を受け取る。命令群と命令キューの実装は、このプロトコルの歴史とともにめざましく成長してきた。最初の頃は、各オブジェクトがひとつの命令キューを持っていたのだ。クライアントが Order リクエスト (命令の型や対象オブジェクトなどの情報を含む) を発行して Outcome レスポンス (命令の結果として期待される詳細) を受け取り、命令が完了したら実際の結果を Result フレームで受け取っていた。

次のバージョンでは Order フレームが Outcome フレームの内容を取り込み (というのも、命令の内容によってはサーバーの入力を必要としないからだ)、Result フレームは完全に廃止された。最新版のプロトコルでは命令キューをオブジェクトから切り離し、Get Order Queue IDs や List of Order Queue IDs、Get Order Queue そして Order Queue といったフレーム

を追加した。これらはメッセージやボード機能と同じように動く[2]。Get Order フレームおよび Remove Order フレーム (どちらも GetWithIDSlot リクエスト) を使えば、クライアントがキューにある命令を取得したり削除したりできる。Insert Order フレームは、Order を運ぶための道具だ。これを用意したのは、別のフレーム Probe Order を考慮したためだ。このフレームを使って、クライアントがローカルで使うための情報を取得することになる。

資源についての説明を取得するのも、通常のアイテム群のパターンと同じだ。まず Get Resource Description IDs リクエストを発行して List of Resource Description IDs レスポンスが取得し、一連の Get Resource Description リクエストで Resource Description レスポンスを得ることになる。

デザインの操作

Thousand Parsec プロトコルにおけるデザインの操作は、四つのサブカテゴリの操作に切り分けられる。カテゴリ、コンポーネント、プロパティ、そしてデザインだ。

カテゴリは、さまざまなデザイン型を区別するためのものだ。最も一般的に使われているデザイン型は、船と兵器の二つである。カテゴリを作るのは単純なことで、名前と説明を指定するだけでよい。Category フレーム自体に含まれるのは、この二つの文字列だけである。ルールセットがカテゴリをデザインストアに追加するときに使うのが Add Category リクエストで、このリクエストが Category フレームを運ぶ。カテゴリ管理のその他の作業を受け持つのは通常のアイテム群のパターンで、Get Category IDs リクエストと List of Category IDs レスポンスを使う。

コンポーネントには、そのデザインを構成するさまざまなパーツやモジュールが含まれている。船体からミサイル、そしてミサイルを据え付ける砲台まであらゆるものが考えられる。コンポーネントは、カテゴリに比べて少し込み入っている。Component フレームに含まれる情報は次のとおりだ。

- コンポーネントの名前と説明。
- コンポーネントが属するカテゴリのリスト。
- Requirements 関数。Thousand Parsec Component Language (TPCL) 形式。
- プロパティとその値のリスト。

コンポーネントに関連づけられた Requirements 関数について、もう少し詳しく説明する。コンポーネントを組み合わせて船や兵器などのオブジェクトを作ることになるので、デザインに追加するときにその妥当性を保証できなければいけない。Requirements 関数は、デザインに追加する各コンポーネントがすでに追加済みの他のコンポーネントのルールを満たすかどうかを検証する。たとえば Missile and Torpedo Wars では、船に Alpha Missile を搭載するには Alpha Missile Tube が必須となる。こういった検証が、クライアント側とサーバー側

[2]実際のところ、別の見方もできる。メッセージやボードが、第二バージョンのプロトコルから派生したものだという見方だ。

```
<prop>
<CategoryIDName>Ships</CategoryIDName>
<rank value="0"/>
<name>Colonise</name>
<displayName>Can Colonise Planets</displayName>
<description>Can the ship colonise planets</description>
<tpclDisplayFunction>
    (lambda (design bits) (let ((n (apply + bits))) (cons n (if (= n 1) "Yes" "No")) ) )
</tpclDisplayFunction>
<tpclRequirementsFunction>
    (lambda (design) (cons #t ""))
</tpclRequirementsFunction>
</prop>
```

図 21.1: プロパティの例

の両方で行われる。そのためには関数全体がプロトコルのフレームに存在しなければいけないし、簡潔な言語 (後述する TPCL) を採用したのもそのためだ。

デザインのプロパティに関する通信は、すべて Property フレームを使って行う。各ルールセットは、ゲーム内で使えるプロパティ群を公開している。一般的なプロパティとしては、一隻の船に搭載できる砲台の最大数や船体の装甲などがある。Component フレームと同様に、Property フレームでも TPCL を使っている。Property フレームに含まれる情報は次のとおりだ。

- プロパティの (表示用の) 名前とその説明。
- プロパティが属するカテゴリのリスト。
- プロパティの (TPCL の識別しとして使える形式の) 名前。
- プロパティのランク。
- Calculate 関数および Requirements 関数。Thousand Parsec Component Language (TPCL) 形式。

プロパティのランクは、依存関係の階層を識別するために利用する。TPCL では、ある関数がこのプロパティのランク以下のプロパティには依存していないことがある。つまり、何かがランク 1 の Armor プロパティとランク 0 の Invisibility プロパティを持っていた場合、Invisibility プロパティは直接 Armor プロパティに依存することができない。ランクを導入したのは、循環依存の問題を回避するためだった。Calculate 関数を使ってそのプロパティの表示方法を定義し、測定方法を差別化する。Missile and Torpedo Wars は、ゲームのデータファイルから XML 形式でプロパティをインポートする。ゲームデータにあるプロパティの例を図 21.1 に示す。

この例では、Ships カテゴリに属するランク 0 のプロパティを扱っている。プロパティの名前は Colonise で、その船が惑星を占領できるかどうかに関わるものだ。TPCL の Calculate 関数 (tpclDisplayFunction) をざっと眺めてみるとわかるとおり、このプロパティの出力は

“Yes” か “No” のいずれかで、船がその機能を持っているかどうかを表す。このようにプロパティを追加していくことで、ルールセットを作る人がゲームをよりきめ細やかに作り込めるようになり、ユーザーにわかりやすい形式でそのプロパティを表示できるようにもなる。

船や兵器などの実際のデザインを作ったり操作したりするときに使うのが、Design フレームとそれに関連するフレーム群だ。いま出回っているすべてのルールセットは、これらを使って既存のコンポーネントプールおよびプロパティプールから船や兵器を組み立てている。プロパティやコンポーネントに関するデザイン時のルールは TPCL の Requirements 関数で処理されているので、デザインを作ることは少しシンプルになる。Design フレームの内容は次のとおりだ。

- デザインの名前とその説明。
- デザインが属するカテゴリのリスト。
- デザインのインスタンス数のカウンタ。
- デザインの所有者。
- コンポーネント ID とそれに対応するカウントのリスト。
- プロパティとそれに対応する表示用文字列のリスト。
- デザインへのフィードバック。

このフレームはその他のフレームとは少し違う。最も注目すべきなのは、デザインというのはゲーム内で誰かが所有するアイテムだという点だ。つまり、個々のデザインには所有者がいることになる。また、個々のデザインは自分自身のインスタンスが作られた回数を覚えている。

サーバー管理

サーバー管理プロトコルの拡張も用意されており、それに対応したサーバーをリモート制御できるようになっている。一般的な使い道は、管理クライアント (シェル風のコマンドラインインターフェイスだったり、あるいは GUI の設定パネルだったりするだろう) 経由でサーバーに接続して設定の変更やその他の保守作業をこなすことだ。しかし、それ以外の使い方もできる。シングルプレイヤーのゲームで裏方の処理を管理することなどだ。

先のセクションで説明したゲームのプロトコルと同様、管理クライアントもまずは接続のネゴシエーション (ゲームで使っているポートとは別のポートを利用する) を行ってから認証に進む。利用するのは Connect リクエストと Login リクエストだ。接続を確立したら、サーバーからログメッセージを受け取ったりサーバーにコマンドを発行したりできるようになる。

ログメッセージをクライアントに送るときに使うのが Log Message フレームだ。この中には、深刻度とメッセージ本文が含まれている。コンテキストに合わせて、クライアント側ではそれを全部表示してもいいし一部だけを表示することもある。あるいは一切メッセージを無視することもある。

サーバーは、Command Update フレームを発行してクライアント側のローカルコマンドセットを更新することもある。サーバー側でサポートするコマンドは、クライアントからの Get Command Description IDs フレームへのレスポンスとして公開する。個々のコマンドの詳細を取得するには、Get Command Description フレームを発行しなければいけない。サーバーはそれに対して Command Description フレームで応答する。

このやりとりは、ゲーム本体で使われていた命令関連のフレーム群とよく似ている (実際、それをもとにして作ったものだ)。これらを使えばコマンドをユーザー側で吟味でき、ネットワークの流量を抑えられる。管理プロトコルができたのは、ゲームプロトコルが既に成熟しきった頃だった。そのため、必要な機能の大半は既にゲームプロトコルにそろっており、ちょっとしたプロトコルライブラリのコードを追加するだけでよかった。

21.3 サポート機能

サーバーの永続化

Thousand Parsec のゲームは、ターン制の戦略型ゲームの多くがそうであるように、それなりに時間をかけてプレイする可能性がある。プレイヤーの日々の生活周期を越えて長時間実行されることもあるので、その間さまざまな理由でサーバープロセスが終了してしまうこともあるだろう。切断されてしまった時点からゲームを再開できるようにするため、Thousand Parsec サーバーは永続化機能を提供している。ゲーム世界全体の状態をデータベースに格納しているのだ。この機能はシングルプレイヤーゲームを保存する仕組みとしても使われており、それについては後で詳しく説明する。

最先端のサーバーである tpserver-cpp では抽象化永続インターフェイスが提供されており、さまざまなデータベースをバックエンドとして使うためのプラグインを作れるシステムもある。本章の執筆時点で tpserver-cpp に同梱されているのは MySQL 用と SQLite 用のプラグインモジュールだ。

抽象クラス Persistence で、サーバーがゲームのさまざまな要素 (『スターエンパイアの世界』で説明したもの) を扱うための仕組みを提供している。保存したり更新したり、現状を取得したりといった機能だ。データベースは、ゲームの状態が変わるたびにサーバーのコードのさまざまな場所から継続的に更新される。どの段階でサーバーが止まったりクラッシュしたりしても、サーバーを再開させればその時点でのすべての情報を復旧できなければいけない。

Thousand Parsec Component Language

Thousand Parsec Component Language (TPCL) は、サーバーとのやりとりなしでクライアントがデザインを作成できるように用意された。プロパティや見た目、デザインの妥当性などに関するフィードバックをその場で得られるようにしたのだ。プレイヤーは、たとえば新た

なクラスの宇宙船を作ったりその構造や推力、機材、防御、武装などをカスタマイズできたりする。

TPCL は Scheme のサブセットである。多少手を加えてはいるものの Scheme R5RS 規格にほぼ沿っており、R5RS 互換のインタプリタならなんでも使える。Scheme を採用した理由はいくつかある。シンプルであること。組み込み言語としての採用事例があったこと、さまざまな他の言語でインタプリタが実装されていること、そして何よりも重要なのがオープンソースプロジェクトであったことだ。言語の使い方に関してもインタプリタの開発法に関しても大量のドキュメントがあった。

次の例は TPCL の `Requirements` 関数で、コンポーネントやプロパティで使われているものだ。これをサーバー側のルールセットに組み込み、ゲームのプロトコルを使ってクライアントと通信する。

```
(lambda (design)
  (if (> (designType.MaxSize design) (designType.Size design))
      (if (= (designType.num-hulls design) 1)
          (cons #t "")
          (cons #f "Ship can only have one hull")
      )
      (cons #f "This many components can't fit into this Hull")
  )
)
```

Scheme になじみのある人なら、ぱっと見ればすぐに理解できるはずだ。ゲームのクライアントとサーバーは、これを使って他のコンポーネントのプロパティ(`MaxSize` や `Size`、そして `Num-Hulls`) をチェックし、このコンポーネントをデザインに追加できるかどうかを調べる。まず最初に、コンポーネントの `Size` がデザインの最大サイズにおさまるかどうかを調べ、次にそのデザインに既に他の船体が存在しないことを確かめる (後者のテストを見れば、この `Requirements` 関数が船体のものであることがわかる)。

BattleXML

戦争では、あらゆる戦いが大切になる。深宇宙での軽武装な偵察機同士の小競り合いであろうが首都上空での主力艦隊同士の最終決戦であろうが、それは変わらない。Thousand Parsec フレームワークでは、戦闘の詳細をルールセットで扱う。戦闘の詳細に関してクライアント側から明示的に何かできる機能は存在しない。通常、プレイヤーに知らされるのは、戦闘が始まったこととその結果を表すメッセージだけであり、それにまつわるオブジェクトへの変更 (破壊された船の削除など) もそこで行われる。プレイヤーが注力するのはもっと上位レベルのことになるが、ルールセットの内部では複雑な戦闘の仕組みが動いており、その戦いの内容を詳しく吟味するのも有益 (少なくとも興味は持ってもらえる) だろう。

ここで使われているのが BattleXML だ。戦闘データは、大きく二つの部分に分かれる。メディアの定義 (利用するグラフィックなどの詳細) と戦いの定義 (戦闘中に実際に発生する出

来事) だ。このデータはバトルビューアが読むことを想定している。現在 Thousand Parsec には二種類のバトルビューアがあり、ひとつは 2D でもうひとつが 3D である。もちろん、戦いの性質はルールセットで定義するものなので、実際に BattleXML のデータを作るのはルールセットのコードの役割となる。

メディアの定義はビューアの特性に結びつけられており。ディレクトリあるいはアーカイブで格納されている。その内容は、XML データおよびそこから参照するグラフィックやモデルのファイルである。データ自体に記述しているのは、船 (あるいはその他のオブジェクト) の型ごとに必要なメディア、炎上や死亡といったアクションに対応するアニメーション、そして武器のメディアや詳細である。ファイルの位置は XML ファイルからの相対パスで参照し、親ディレクトリを参照することはできない。

戦いの定義は、ビューアやメディアからは独立している。まず、戦闘開始時の両陣営のエンティティについての ID と情報 (名前や説明、そして型など) を記述する。次に、戦闘の各ラウンドについて記述する。オブジェクトの移動や武器の使用 (攻撃元と攻撃先)、オブジェクトへのダメージ、オブジェクトの死亡、ログメッセージなどである。各ラウンドをどの程度詳細に記述するかはルールセットが決めることだ。

メタサーバー

公開されている Thousand Parsec サーバーを探すというのは、まるで深宇宙でステルス偵察機を探すようなものだ。場所を知っていなければ、成功する見込みは薄い。幸いにも、公開サーバーはメタサーバーにアナウンスできるようになっている。その場所を中央ハブとしてアナウンスすることで、プレイヤーから見つけられるようにできるのだ。

現在の実装は `metaserver-lite` という PHP スクリプトで、Thousand Parsec のウェブサイトなどいくつかの場所で公開されている。サポートするサーバーは、HTTP リクエストを送って update アクションを指定し、型や場所 (プロトコル、ホスト、ポート)、ルールセット、プレイヤー数、オブジェクト数、管理者などの情報を伝える。サーバー一覧の情報は一定期間 (デフォルトでは 10 分) でタイムアウトとなるので、サーバーは定期的にメタサーバーを更新することが求められる。

このスクリプトに何もアクションを指定せずに呼ぶと、サーバーの一覧とその詳細をウェブサイトに組み込むことができる。そしてそこに、クリッカブルな URL(通常は `tp://`スキームとなる) を含めることができる。あるいは、badge アクションを指定すれば、サーバー一覧をコンパクトな「バッジ」形式で表示できる。

クライアントからメタサーバーに対して get アクションのリクエストを発行すれば、利用可能なサーバーの一覧を取得できる。このときメタサーバーがクライアントに返すのは、一覧の各サーバーについての `Game` フレームである。`tpclient-pywx` では、この結果一覧を初期接続ウィンドウのサーバーブラウザに表示する。

シングルプレイヤーモード

Thousand Parsec は、ネットワーク対応のマルチプレイヤーゲームとして作られたものだ。しかし別に、ローカルサーバーを立ち上げた一人のプレイヤーがそこに数台の AI クライアントを接続し、単独でその世界に飛び込んで征服しようとしたっていっこうにかまわない。そんなときのための標準的なメタデータや機能も用意されており、GUI のウィザードを実行するかシナリオファイルをダブルクリックするだけで準備できる。

この機能の中核となるのが、各コンポーネント (サーバー、AI クライアント、ルールセット) の機能やプロパティにまつわるメタデータ用のフォーマットを指定した XML DTD である。コンポーネントのパッケージにはこういった XML ファイルが同梱されており、最終的にこれらのメタデータがすべて連想配列にとりまとめられる。この連想配列は大きく二つの部分に分かれている。サーバーと AI クライアントだ。サーバーのメタデータの中には、一般的にはいくつかのルールセットのメタデータが入っている。ルールセットの実装は複数のサーバーを対象にしたものだがその細かい設定はサーバーによって異なるかもしれないので、実装ごとに個別のメタデータが必要になるのだ。これらのコンポーネントの各エントリには、次の情報が含まれている。

- 説明のデータ。短い名前や長い名前そして説明文など。
- インストールされているコンポーネントのバージョン、そしてそのバージョンで互換性があるのはどのバージョン以降で作ったセーブデータか。
- 渡すコマンド文字列 (もし受け付けているなら) および強制パラメータ。
- プレイヤーが設定できるパラメータ群。

強制パラメータはプレイヤーが設定することはできない。一般的には、コンポーネントがシングルプレイヤーモードに合わせて適切に機能するよう設定されている。プレイヤーが設定ｄけいるパラメータはそれぞれ独自のフォーマットで、たとえば名前や説明、データ型、デフォルト、値の範囲、コマンド文字列に付加する書式文字列などを指定する。

特殊な場合 (そのルールセット固有のクライアント用のプリセット設定でゲームをする場合など) もあり得るが、一般的なシングルプレイヤーゲームの手順は互換性のあるコンポーネント群を選ぶところから始まる。クライアントの選択は暗黙のうちに行われる。というのもプレイヤーは既に、ゲームをプレイするためにひとつのクライアントを立ち上げているからである。きちんと設計されたクライアントが、ユーザー中心のワークフローに沿って残りを準備する。次に選ばなければいけないのは当然、ルールセットだ。そこで、ルールセットの一覧をプレイヤーに表示する。この時点では、サーバーの詳細を気にする必要はない。選択したルールセットが複数のサーバーで実装されている場合 (おそらくめったにないことだろう) は、そのうちのいずれかを選ぶようプレイヤーに問いかける。そうでなければ、適切なサーバーが自動的に選ばれる。次にプレイヤーは、ルールセットとサーバーのオプションを設定するよう指示を受ける。あらかじめメタデータからのデフォルト値が設定されているものだ。最後に、もし互換性のある AI クライアントがインストールされていれば、それらの対戦相手の設定を行う。

ゲームの設定をするとともに、クライアントはローカルサーバーを立ち上げる。メタデータから取得したコマンド文字列の情報を使って、適切なパラメータ (ルールセットやそのパラメータ、その他サーバーに関する設定項目) を設定する。サーバーが立ち上がって接続を受け付ける状態になったら、先ほど説明した管理プロトコルの拡張を使って AI クライアントも同様に立ち上げる。そして、それらもきちんとゲームに接続できたことを確かめる。すべてがうまくいけば、クライアントが (オンラインゲームのときと同じように) サーバーに接続し、プレイヤーはゲームの世界で動き出せるようになる。

シングルプレイヤーモードのもうひとつの (そしてとても重要な) 機能は、ゲームをセーブしたりロードしたりできることだ。そしてそれと同じくらい大切なのが、すぐにプレイできる状態のシナリオを読み込む機能である。このときのセーブデータ (必須ではないが、おそらくひとつのファイルになるだろう) に格納される内容は、シングルプレイヤーゲームの設定データやゲーム自体の永続データである。すべてのコンポーネントについて互換性のあるバージョンがインストールされてさえいれば、セーブしたゲームやシナリオを自動的に立ち上げることができる。特にシナリオに関しては、ワンクリックでゲームを始められるような仕組みを提供している。今のところ Thousand Parsec には専用のシナリオエディタはないし、エディットモードつきのクライアントも存在しない。しかし、何らかの手段を用意して通常のルールセットとの機能とは別に永続データを作れるようにする考えはある。その一貫性や互換性も検証できるようにするつもりだ。

ここまでに説明した内容は、あくまでも抽象レベルのものだ。実装レベルで見ると、Thousand Parsec プロジェクトの Python クライアントヘルパーライブラリである `libtpclient-py` が今のところ唯一シングルプレイヤーモードのすべての仕組みを理解できるものである。このライブラリには `SinglePlayerGame` クラスがある。これは自動的にインスタンス化されてシステム上のすべてのシングルプレイヤー用メタデータを収集する (当然、システム上のどこに XML ファイルをインストールするかについてはプラットフォームごとにガイドラインがある)。クライアントからは、利用可能なコンポーネント (サーバーやルールセット、AI クライアント、そして Python の連想配列であるディクショナリに格納されているパラメータなど) に関するさまざまな情報を問い合わせられるようになる。先ほど説明したゲームの組み立て手順に照らし合わせると、一般的なクライアントの動きは次のようになる。

1. 利用できるルールセットの一覧を `SinglePlayerGame.rulesets` で問い合わせ、選択したルールセットのオブジェクトを `SinglePlayerGame.rname` で設定する。
2. 選択したルールセットを実装するサーバーを `SinglePlayerGame.list_servers_with_ruleset` で問い合わせ、複数存在する場合はその中からユーザーに選ばせる。そして、選んだサーバーのオブジェクトを `SinglePlayerGame.sname` で設定する。
3. そのサーバーとルールセット用のパラメータセットをそれぞれ `SinglePlayerGame.list_rparams` `SinglePlayerGame.list_sparams` で取得し、プレイヤーに設定させる。
4. そのルールセットに対応している AI クライアントを `SinglePlayerGame.list_aiclients_with_ruleset` で探し、`SinglePlayerGame.list_aiparams` で取得したパラメータを使ってプレイヤーに設定させる。

5. `SinglePlayerGame.start` を呼んでゲームを始める。これは、成功した場合に接続先の TCP/IP ポートを返す。
6. 最終的にゲームを終了させる (そして、立ち上がっているサーバーと AI クライアントのプロセスを終了させる) には `SinglePlayerGame.stop` を呼ぶ。

Thousand Parsec のフラッグシップクライアントである `tpclient-pywx` には使いやすいウィザードが組み込まれており、セーブしたゲームやシナリオファイルを読み込むところからこれらの手順を進めてくれる。このウィザードに組み込まれたユーザー主導のワークフローはまさに、オープンソースプロジェクトの開発の成果の一例と言えるだろう。開発者が最初に用意したのはもっと違う手順で、実際の内部的な動きにより近いものだった。コミュニティでの議論や共同開発などのおかげで、それがよりユーザーに使いやすいものになったのだ。

最後に、保存されたゲームやシナリオについては、現在は `tpserver-cpp` で実装されており、`libtpclient-py` のサポート機能や `tpclient-pywx` のインターフェイスを使っている。SQLite を使う永続化モジュールでこれを実現した。SQLite はパブリックドメインなオープンソースの RDBMS で、外部のプロセスを必要としないしデータベースを単一のファイルとして格納できる。サーバーの設定を強制パラメータ経由で行うときには、もし使えれば SQLite の永続化モジュールを利用する。そして、データベースのファイル (テンポラリディレクトリにある) はゲームの実行中常に更新される。プレイヤーがゲームのセーブをするときにはデータベースファイルを指定の場所にコピーし、シングルプレイヤーの設定データを格納するための特別なテーブルを追加する。これをあとでどうやって読み込むのかは、リーダーがよく知っているはずだ。

21.4 教訓

大規模な Thousand Parsec フレームワークを作って成長させていく過程を振り返ると、開発者達が設計に関する決断を下す場面が数多く存在した。最初のコアデベロッパー (Tim Ansell と Lee Begg) はフレームワークをスクラッチから構築し、私たちが同様のプロジェクトを始めるにあたっていろんな提案をしてくれた。

うまくいったこと

Thousand Parsec の開発における大切なポイントは、フレームワークのサブセットを定義および構築してからそれを実装すると決めたことだった。イテレーティブかつインクリメンタルな設計プロセスを使ったことでフレームワークは組織的に成長し、新機能もシームレスに追加できるようになった。その流れで Thousand Parsec プロトコルもバージョンで管理することになり、これもフレームワークの成功に大きく貢献した。プロトコルをバージョンで管理したおかげで、フレームワークを日々成長させてゲーム用のメソッドもどんどん追加できるようになったのだ。

広範囲の及ぶフレームワークを開発するときに重要なのは、イテレーションごとの短期的なゴールを定めてそこに向かっていくことだ。数週間レベルの短いイテレーションでマイナーリリースをするようにすると、プロジェクトの進み具合も速くなるし、すぐに反応が得られるようになる。もうひとつうまくいったことがあって、それは実装にクライアント・サーバーモデルを採用したことだ。そのおかげで、クライアントの開発をゲームのロジックとは切り離して考えられるようになった。ゲームのロジックとクライアントソフトウェアを分離したことが Thousand Parsec の大成功につながった。

うまくいかなかったこと

Thousand Parsec フレームワークの最大の失敗は、バイナリプロトコルを使ったことだ。だいたい想像できるだろうが、バイナリプロトコルのデバッグは決して楽しい作業じゃない。デバッグにかかる時間もどんどん伸びてしまう。これから何かを実装しようとする人たちには、バイナリプロトコルは使わないようお勧めしたい。さらに、このプロトコルはどんどん成長して、必要以上に柔軟性を持ちすぎるようになってしまった。何かのプロトコルを作る際には、必要最小限の基本的な機能だけを実装することが大切だ。

私たちの開発のイテレーションは、たまに長くなりすぎることもあった。巨大なフレームワークの開発をオープンソースの開発スケジュールで管理するときに大切なのは、追加された機能の小さなサブセットを各イテレーションで扱うようにして、開発をうまく流れるようにしていくことだ。

結論

軌道上の造船所で小舟に乗って巨大戦艦のプロトタイプの船体を調査するがごとく、本章では Thousand Parsec のアーキテクチャの詳細をいろいろ見てきた。全体的な設計指針として、柔軟性と拡張性を重視するということが最初からずっと開発者達の念頭にあった。ここまでの流れを振り返っても明らかなように、オープンソースのエコシステムに身を置いて仲間たちのいろいろなアイデアや観点を取り入れたからこそこのフレームワークができあがったのだ。さまざまな可能性を秘めつつ、機能的にも優れていてきちんとまとまっている。これは非常に野心的なプロジェクトなので、オープンソースの世界の仲間たちと同様に、私たちにもまだまだやるべきことが残っている。Thousand Parsec は今後も成長を続けるだろうし、その機能を拡張し続けるだろう。それを使って、新たなゲーム・より複雑なゲームも開発されるはずだ。結局のところ、1000 パーセクの旅と言ってもはじめの一歩は小さなものなんだ。

第22章

Violet

Cay Horstmann

2002 年に、学部生向けにオブジェクト指向設計やパターンの教科書を執筆したことがある [Hor05]。きっと他の多くの人もそうだろうが、私がその本を書いたのは正規の教育カリキュラムに不満があったからである。よくありがちなカリキュラムでは、計算機科学の学生が最初に学ぶのは単一のクラスを設計する方法だ。そして、その先はオブジェクト指向設計に関するトレーニングを一切受けずに上級レベルのソフトウェア工学コースに進んでしまう。そのコースで学ぶ内容といえば、UML だとかデザインパターンだとかをほんの数週間で詰め込むだけ。そんなことしても、学生を混乱させてしまうだけのことだ。私が書いた教科書は、学生が半年かけて学ぶコースを想定したものだ。Java のプログラミングとデータ構造に関する基礎知識 (Java ベースのコースで 1 年程度学んでいる程度の知識) があることを前提としている。扱う内容は、オブジェクト指向設計の原則やデザインパターンを、よりなじみやすいシチュエーションにまとめたものだ。たとえば Decorator パターンを紹介するときには Swing の `JScrollPane` を使っている。この例のほうが、よくありがちな Java のストリームを使った例より記憶に残りやすいだろう。

図 22.1: Violet のオブジェクト図

執筆にあたって、UML のサブセットが必要になった。クラス図、シーケンス図、そしてオブジェクト図の変種で Java のオブジェクト参照を示すものだ (図 22.1)。また、学生が図を自

分で描けるようにもしたかった。しかし、Rational Rose のような商用製品は、高価だし覚えにくい [Shu05]。一方オープンソースの製品も、その当時は機能的な制限が多かったしバグも多くて使いづらかった [1]。特に、ArgoUML のシーケンス図はまったく使い物にならなかった。

そこで、シンプルなエディタを自前で作ってしまうことにした。このエディタは「学生にとって有用であること」「拡張可能なフレームワークとして、学生がその内容を学んで手を加えられること」を目指した。その結果として誕生したのが Violet である。

22.1 Violet とは

Violet は軽量な UML エディタである。学生や教師、そして書籍の執筆者など、シンプルな UML 図を手早く描きたい人たちに向けて作った。使い方は簡単なので、すぐに使いこなせるようになる。対応している図は、クラス図とシーケンス図、ステート図、オブジェクト図、そしてユースケース図である (その後、他の開発者たちの強力でその他の図にも対応するようになった)。オープンソースのソフトウェアであり、クロスプラットフォームで動く。Violet の中心となるのが、シンプルながらも柔軟性のあるグラフフレームワークで、これは Java 2D グラフィック API をフル活用したものである。

Violet のユーザーインターフェイスは、敢えてシンプルにした。属性やメソッドを入力するために何段階ものダイアログをたどっていく必要はない。単にテキストフィールドに直接入力すればいいだけだ。マウスを数回クリックするだけで、見栄えが良くて使いやすい図をあっという間に作れる。

Violet は業務レベルでの使用に耐えうる UML プログラムを目指しているわけではない。Violet に搭載されて**いない**機能を以下にまとめた。

- Violet には、UML 図からソースコードを生成したりソースコードから UML 図を生成したりする機能はない。
- Violet はモデルのセマンティックチェックをしない。Violet を使って、意味的に矛盾する図を描くこともできる。
- Violet は、他の UML ツールに取り込めるような形式のファイルを生成する機能がない。また、他のツールが書き出したファイルを読み込む機能もない。
- Violet には、図のレイアウトを自動的に調整する機能はない。ただ単に「グリッドに揃える」機能があるだけである。

(この中のどれかに対応してみるっていうのも、学生向けのプロジェクトとしてよさそうだね)。

Violet は、ある種のソフトウェアデザイナーたちから熱狂的な支持を得た。紙ナプキンに手書きするよりはましなツールが欲しいけど、別に業務レベルの UML ツールみたいなのはいらないといった人たちだ。そんな人たちの支持を受けて、私はそのコードを SourceForge

[1] その当時の私は Diomidis Spinellis が作ったすばらしい UMLGraph [Spi03] の存在を知らなかった。このプログラムによる図は、ありがちなマウスクリックによるインターフェイスではなくテキストによる宣言で指定するものだった。

に GNU General Public License で公開した。2005 年には Alexandre de Pellegrin がプロジェクトに合流し、Eclipse プラグインやインターフェイスの改善に貢献してくれた。彼はその後もアーキテクチャの改良を続け、今ではこのプロジェクトのメインメンテナーになっている。

本章では、Violet を最初に作ったときのアーキテクチャに関する決定だけでなく、その後どのように進化したのかも解説する。本章の一部はグラフの描画に特化した内容になっているが、その他の部分 (JavaBeans のプロパティや永続化の利用、Java WebStart、そしてプラグイン機構など) はより多くの人の興味をひく内容になっているはずだ。

22.2 グラフフレームワーク

Violet のベースになっているのは汎用的なグラフ編集フレームワークで、このフレームワークは任意の形状のノードやエッジを表示したり編集したりできる。Violet UML エディタにはクラスやオブジェクト、アクティベーションバー (シーケンス図用) などのノードがあり、UML 図に登場するさまざまな形状のエッジもある。グラフフレームワークの利用例としては、ER 図や路線図などの表示も考えられる。

図 22.2: エディタフレームワークのシンプルな例

このフレームワークを解説する前に、非常にシンプルなグラフを描くだけのエディタを考えてみよう。白と黒の円形のノード、そして直線のエッジだけを使えるようなものだ (図 22.2)。

SimpleGraph クラスはノードとエッジの型を表すプロトタイプオブジェクトを指定するもので、プロトタイプパターンに沿っている。

```
public class SimpleGraph extends AbstractGraph
{
  public Node[] getNodePrototypes()
  {
    return new Node[]
    {
      new CircleNode(Color.BLACK),
      new CircleNode(Color.WHITE)
    };
  }
  public Edge[] getEdgePrototypes()
  {
    return new Edge[]
    {
      new LineEdge()
    };
  }
}
```

プロトタイプオブジェクトを使って、ノードやエッジのボタンを図 22.2 の上部に描画する。ユーザーがノードやエッジのインスタンスを追加するたびに、プロトタイプオブジェクトがクローンされる。Node と Edge はインターフェイスで、それぞれ次のようなメソッドを持つ。

- どちらのインターフェイスにも getShape メソッドがあり、これはそのノードあるいはエッジの形状を表す Java2D Shape オブジェクトを返す。
- Edge インターフェイスには、エッジの先頭と最後にノードを追加するためのメソッドがある。
- Node インターフェイスの getConnectionPoint メソッドは、ノードの境界上での最適な接続ポイントを計算する (図 22.3 を参照)。
- Edge インターフェイスの getConnectionPoints メソッドは、そのエッジの二つの端点を作る。このメソッドが必要なのは、現在選択中のエッジをマークする「grabber」を描画するときである。
- ノードは子供を持つことができ、子供は親とともに移動する。子供たちを列挙したり操作したりするための、一連のメソッドが用意されている。

便利なクラスである AbstractNode と AbstractEdge にはこれらのメソッドの多くが実装されている。そして RectangularNode や SegmentedLineEdge では、それぞれタイトル文字列つきの矩形ノードや破線エッジの完全な実装を提供している。

ここで扱うシンプルなグラフエディタの場合は、CircleNode や LineEdge といったサブクラスを用意する必要があるだろう。これらのクラスが draw メソッドや contains メソッドを提供し、さらにノードの境界の形状を示す getConnectionPoint メソッドも用意する。そのコードを次に示す。図 22.4 はこれらのクラスのクラス図である (もちろん Violet で描いたものだ)。

図 22.3: ノードの境界上での接続ポイントを探す

```
public class CircleNode extends AbstractNode
{
  public CircleNode(Color aColor)
  {
    size = DEFAULT_SIZE;
    x = 0;
    y = 0;
    color = aColor;
  }

  public void draw(Graphics2D g2)
  {
    Ellipse2D circle = new Ellipse2D.Double(x, y, size, size);
    Color oldColor = g2.getColor();
    g2.setColor(color);
    g2.fill(circle);
    g2.setColor(oldColor);
    g2.draw(circle);
  }

  public boolean contains(Point2D p)
  {
    Ellipse2D circle = new Ellipse2D.Double(x, y, size, size);
    return circle.contains(p);
  }

  public Point2D getConnectionPoint(Point2D other)
  {
    double centerX = x + size / 2;
    double centerY = y + size / 2;
    double dx = other.getX() - centerX;
    double dy = other.getY() - centerY;
    double distance = Math.sqrt(dx * dx + dy * dy);
    if (distance == 0) return other;
    else return new Point2D.Double(
      centerX + dx * (size / 2) / distance,
      centerY + dy * (size / 2) / distance);
```

```
  }

  private double x, y, size, color;
  private static final int DEFAULT_SIZE = 20;
}

public class LineEdge extends AbstractEdge
{
  public void draw(Graphics2D g2)
  { g2.draw(getConnectionPoints()); }

  public boolean contains(Point2D aPoint)
  {
    final double MAX_DIST = 2;
    return getConnectionPoints().ptSegDist(aPoint) < MAX_DIST;
  }
}
```

図 22.4: Simple Graph のクラス図

ここまでをまとめよう。Violet では、グラフエディタを作るためのシンプルなフレームワークを提供している。エディタのインスタンスを取得するには、ノードやエッジのクラス群を

定義してグラフ内でのメソッドを用意し、ノードオブジェクトやエッジオブジェクトのプロトタイプを作れるようにする。

もちろんグラフ用のフレームワークは他にも存在する。JGraph [Ald02] や JUNG[2] などだ。でも、これらのフレームワークはかなり複雑だし、あくまでもグラフを描画するためのフレームワークであってグラフ描画アプリケーションを作るためのフレームワークではない。

22.3　JavaBeans のプロパティの利用

クライアントサイドでの Java が花盛りだった頃に、JavaBeans の仕様が策定された。ビジュアル GUI ビルダー環境で、GUI コンポーネントを編集するポータブルな仕組みを提供するためだ。サードパーティの GUI コンポーネントを GUI ビルダーにドロップして、標準のボタンやテキストボックスなどと同様にプロパティを設定できるようにするというのが狙いだった。

Java では、プロパティという仕組みはネイティブで提供されていない。JavaBeans のプロパティはゲッターメソッドとセッターメソッドのペアになっていたり、付随する BeanInfo クラスで指定されていたりする。さらに、**プロパティエディタ**を指定してビジュアルにプロパティの値を編集できる。JDK にも基本的なプロパティエディタが含まれている。`java.awt.Color` がその一例だ。

Violet フレームワークは JavaBeans の仕様をフル活用している。たとえば `CircleNode` クラスは、次の二つのメソッドを用意すれば color プロパティを公開できる。

```
public void setColor(Color newValue)
public Color getColor()
```

たったこれだけだ。グラフエディタは、これで円ノードの色を編集できるようになった (図 22.5)。

22.4　長期持続性

どんなエディタプログラムでもそうだが、Violet はユーザーが作った図をファイルに保存したり図をファイルから読み込めるようにしたりしないといけない。そこで私は、XMI の仕様[3] を調べてみた。これは UML のモデル用のデータ交換フォーマットとして作られたものだ。その結果わかったことは、こいつはやたら面倒くさくて紛らわしくて、処理しづらいものだということだった。そう思ったのは私だけではないと思う。XMI を使うと、極めてシンプルなモデルであっても相互運用性に乏しくなるという評価がある [PGL$^+$05]。

単純に Java のシリアライズ機能を使おうかとも考えた。しかし、旧バージョンでシリアライズしたオブジェクトを読み込むのが難しい。シリアライズの実装は変化し続けているからである。同じ問題を心配していた JavaBeans アーキテクトたちは、標準化した XML フォー

[2] http://jung.sourceforge.net
[3] http://www.omg.org/technology/documents/formal/xmi.htm

図 22.5: デフォルトの JavaBeans Color Editor による円の色の編集

マットで長期持続性を確保しようとした [4]。Java オブジェクト (Violet の場合なら UML 図) を、それを構築したり修正したりするための一連の文で表すというものだ。例を示そう。

```
<?xml version="1.0" encoding="UTF-8"?>
<java version="1.0" class="java.beans.XMLDecoder">
 <object class="com.horstmann.violet.ClassDiagramGraph">
  <void method="addNode">
   <object id="ClassNode0" class="com.horstmann.violet.ClassNode">
    <void property="name">...</void>
   </object>
   <object class="java.awt.geom.Point2D$Double">
    <double>200.0</double>
    <double>60.0</double>
   </object>
  </void>
  <void method="addNode">
   <object id="ClassNode1" class="com.horstmann.violet.ClassNode">
    <void property="name">...</void>
   </object>
   <object class="java.awt.geom.Point2D$Double">
    <double>200.0</double>
    <double>210.0</double>
   </object>
```

[4] http://jcp.org/en/jsr/detail?id=57

```
  </void>
  <void method="connect">
   <object class="com.horstmann.violet.ClassRelationshipEdge">
    <void property="endArrowHead">
     <object class="com.horstmann.violet.ArrowHead" field="TRIANGLE"/>
    </void>
   </object>
   <object idref="ClassNode0"/>
   <object idref="ClassNode1"/>
  </void>
 </object>
</java>
```

XMLDecoder クラスがこのファイルを読み込むと、こんな文を実行する (単純にするために、パッケージ名は省略した)。

```
ClassDiagramGraph obj1 = new ClassDiagramGraph();
ClassNode ClassNode0 = new ClassNode();
ClassNode0.setName(...);
obj1.addNode(ClassNode0, new Point2D.Double(200, 60));
ClassNode ClassNode1 = new ClassNode();
ClassNode1.setName(...);
obj1.addNode(ClassNode1, new Point2D.Double(200, 60));
ClassRelationShipEdge obj2 = new ClassRelationShipEdge();
obj2.setEndArrowHead(ArrowHead.TRIANGLE);
obj1.connect(obj2, ClassNode0, ClassNode1);
```

コンストラクタやプロパティそしてメソッドといった構文が変わらない限り、プログラムのバージョンが新しくなっても旧バージョンで保存したファイルを読み込める。

こんなファイルを作るのは極めて簡単なことだ。エンコーダーが自動的にオブジェクトのプロパティを列挙し、デフォルトと違う値が設定されているプロパティについてその値を設定するセッターを書き出せばよい。ほとんどの基本データ型は Java プラットフォームが処理してくれる。しかし、Point2D や Line2D そして Rectangle2D については特別なハンドラを用意しなければいけなかった。最も大切なのは、グラフが一連の addNode メソッドと connect メソッドにシリアライズできるということをエンコーダーが知っておく必要があるということだ。

```
encoder.setPersistenceDelegate(Graph.class, new DefaultPersistenceDelegate()
{
  protected void initialize(Class<?> type, Object oldInstance,
    Object newInstance, Encoder out)
  {
    super.initialize(type, oldInstance, newInstance, out);
    AbstractGraph g = (AbstractGraph) oldInstance;
    for (Node n : g.getNodes())
      out.writeStatement(new Statement(oldInstance, "addNode", new Object[]
      {
        n,
```

```
        n.getLocation()
      }));
    for (Edge e : g.getEdges())
      out.writeStatement(new Statement(oldInstance, "connect", new Object[]
      {
        e, e.getStart(), e.getEnd()
      }));
  }
});
```

エンコーダーをきちんと設定してしまえば、グラフを保存するのは簡単で、単にこのようにするだけだ。

```
encoder.writeObject(graph);
```

デコーダーは単純に文を実行するだけなので、設定は何もいらない。グラフを読むにはこうするだけでいい。

```
Graph graph = (Graph) decoder.readObject();
```

この手法は大成功を収め、これまでのさまざまなバージョンの Violet でうまく動いてきたが、唯一の例外があった。最近施したリファクタリングでパッケージ名を変更してしまい、過去との互換性が崩れてしまったのだ。新しいパッケージ構成からは外れてしまうが、旧パッケージのクラスをそのまま残しておくという選択肢もあった。そうはせず、メンテナーが XML トランスレーターを用意することにした。旧バージョンのファイルを読み込むときにパッケージ名を自動的に書き換えるようにしたのだ。

22.5 Java Web Start

Java Web Start を使えば、アプリケーションを Web ブラウザから起動することができる。JNLP ファイルをデプロイしておけばこれがブラウザ内でヘルパーアプリケーションを起動し、Java のプログラムをダウンロードして実行してくれる。アプリケーションにデジタル署名を施すこともできる。その場合、ユーザーは証明書を受け入れないとアプリケーションを実行できない。署名なしの場合は、プログラムはサンドボックス内で実行される。このサンドボックスは、アプレットのサンドボックスより若干制限の緩いものだ。

私は、デジタル証明書の妥当性やそのセキュリティ的に意味するところを一般の利用者が判断できるとは思っていないし、またそんな判断をさせるべきではないと考える。Java プラットフォームの強みの一つがそのセキュリティなのだから、その強みを利用することが重要だろう。

Java Web Start のサンドボックスは、一般の利用者が作業を進めるのには十分なほど強力で、ファイルの読み込みや保存もできるし印刷もできる。利用者から見たときに、これらの操作は安全かつ便利に利用できる。アプリケーションがローカルのファイルシステムにアク

セスしようとしたときには確認のメッセージが出るので、それを見てファイルの読み書きをさせるかどうかを選べるようになる。アプリケーションは単にストリームオブジェクトを受け取るだけであり、ファイル選択の間にも実際にファイルシステムを見ることはない。

開発者にとっては少し面倒になる。アプリケーションを Web Start の配下で実行させるには FileOpenService や FileSaveService を使うコードを書かなければいけないからである。さらに面倒なのは、アプリケーションが Web Start 経由で起動されたのかどうかを調べるための Web Start API が存在しないということだ。

同様に、ユーザーの設定項目を保存するにも二通りの実装方法がある。アプリケーションを普通に実行している場合は Java preferences API を使うし、Web Start 配下で動いている場合は Web Start preferences service を使うことになる。一方印刷については、アプリケーションのプログラマー側からはまったく同じように扱える。

Violet ではこれらのサービスの上にシンプルな抽象化層を提供し、アプリケーションのプログラマーの作業を単純化させている。たとえばファイルを開くには次のようにすればよい。

```
FileService service = FileService.getInstance(initialDirectory);
  // Web Starth 配下で動いているかどうかを検出する
FileService.Open open = fileService.open(defaultDirectory, defaultName,
  extensionFilter);
InputStream in = open.getInputStream();
String title = open.getName();
```

FileService.Open インターフェイスを実装しているクラスは二つで、JFileChooser のラッパーかあるいは JNLP の FileOpenService である。

この便利な仕組みは JNLP API そのものの機能だというわけではない。しかしこの API はこれまでほとんど顧みられることがなく、無視され続けてきた。大半のプロジェクトは自己署名の証明書を使って Web Start アプリケーションを公開しており、利用者側のセキュリティを無視している。これは恥ずべきことだ。オープンソースの開発者は、JNLP のサンドボックスを受け入れてリスクなくプログラムを動かせるようにするべきだ。

22.6 Java 2D

Violet は Java2D ライブラリを徹底的に使いまくっている。このライブラリは、Java API の中でも認知度の低いもののひとつだ。すべてのノードやエッジには getShape メソッドがあり、java.awt.Shape を返すようになっている。これは、Java2D のすべての図形の共通インターフェイスだ。このインターフェイスを実装して、矩形や円やパスそしてそれらの和集合や積集合、差分などができあがる。GeneralPath クラスは、任意の直線および二次曲線・三次曲線を合成した図形 (直線の矢印や曲線の矢印など) を作るのに有用なクラスだ。

Java2D API の柔軟性を理解するために、次のコードを見てみよう。これは AbstractNode.draw メソッドで影を描画するためのコードだ。

```
Shape shape = getShape();
if (shape == null) return;
g2.translate(SHADOW_GAP, SHADOW_GAP);
g2.setColor(SHADOW_COLOR);
g2.fill(shape);
g2.translate(-SHADOW_GAP, -SHADOW_GAP);
g2.setColor(BACKGROUND_COLOR);
g2.fill(shape);
```

ほんの数行のコードで、あらゆる図形に影をつけることができる。開発者が後から追加した図形でも同じだ。

もちろん、Violet が保存する画像はビットマップ形式だ。javax.imageio パッケージがサポートするあらゆるフォーマット、つまり GIF や PNG そして JPEG などに対応している。出版社からベクター形式の画像を求められたときには、Java 2D ライブラリを使う別のメリットを説明した。PostScript プリンタに印字するときに、Java2D での操作は PostScript のベクター描画操作に変換されるのだ。ファイルに印刷すれば、その結果を ps2eps などに食わせて Adobe Illustrator あるいは Inkscape にインポートできるようになる。そのコード例を示す。ここで comp は Swing コンポーネントであり、その paintComponent メソッドがグラフを描画する。

```
DocFlavor flavor = DocFlavor.SERVICE_FORMATTED.PRINTABLE;
String mimeType = "application/postscript";
StreamPrintServiceFactory[] factories;
StreamPrintServiceFactory.lookupStreamPrintServiceFactories(flavor, mimeType);
FileOutputStream out = new FileOutputStream(fileName);
PrintService service = factories[0].getPrintService(out);
SimpleDoc doc = new SimpleDoc(new Printable() {
  public int print(Graphics g, PageFormat pf, int page) {
      if (page >= 1) return Printable.NO_SUCH_PAGE;
      else {
        double sf1 = pf.getImageableWidth() / (comp.getWidth() + 1);
        double sf2 = pf.getImageableHeight() / (comp.getHeight() + 1);
        double s = Math.min(sf1, sf2);
        Graphics2D g2 = (Graphics2D) g;
        g2.translate((pf.getWidth() - pf.getImageableWidth()) / 2,
            (pf.getHeight() - pf.getImageableHeight()) / 2);
        g2.scale(s, s);

        comp.paint(g);
        return Printable.PAGE_EXISTS;
      }
  }
}, flavor, null);
DocPrintJob job = service.createPrintJob();
PrintRequestAttributeSet attributes = new HashPrintRequestAttributeSet();
job.print(doc, attributes);
```

当初私が気にかけていたのは、汎用的な図形クラスを使えばパフォーマンスが落ちるのでは

ないかということだった。しかし、実際はそうでもないことがわかった。クリッピングを使って、現在のビューポイントの更新に必要な図形だけを処理するようにすればよかったのだ。

22.7 Swing アプリケーションのフレームワークがない

たいていの GUI フレームワークでは、メニューやツールバーそしてステータスバーなどを使ってドキュメント群を扱うアプリケーションを作ることを想定している。しかし、Java API にはそんな仕組みはなかった。かつて JSR 296[5] で Swing アプリケーション用の基本的なフレームワークを提供しようとしていたが、今では活動していない。そこで、Swing アプリケーションの作者は二つの選択肢のいずれかを選ぶことになる。車輪の再発明を大量に繰り返すか、あるいはサードパーティのフレームワークを使うかだ。Violet を書き始めた頃、アプリケーションフレームワークとして最も有力な選択肢は Eclipse と NetBeans だった。しかし当時は、どちらも重量級過ぎるように感じられた (最近は他の選択肢も増えた。中には JSR 296 からのフォークである GUTS[6] のようなものもある)。そのため、Violet では車輪の再発明をする道を選ばざるを得なかった。メニューや内部ウィンドウを扱う仕組みを自前で用意したのだ。

Violet では、メニューの項目はプロパティファイルで次のように指定する。

```
file.save.text=Save
file.save.mnemonic=S
file.save.accelerator=ctrl S
file.save.icon=/icons/16x16/save.png
```

ユーティリティメソッドが、メニューの項目をプレフィックス (ここでは `file.save`) から作る。サフィックスである `.text` や `.mnemonic` などは、今風に言うと「設定より規約」というやつだ。リソースファイルを使ってこれらの設定をするという方法は、API を呼んでメニューを作る方法よりも遥かに優れている。ローカライゼーションが簡単にできるからだ。私は別のオープンソースプロジェクトでもこの仕組みを流用した。ハイスクールでのコンピュータサイエンス教育用環境である GridWorld[7] だ。

Violet のようなアプリケーションでは、複数の「ドキュメント」を開くことができる。そしてそれぞれのドキュメントにグラフが含まれる。Violet を最初に書いた頃は、マルチドキュメントインターフェイス (MDI) がまだ主流だった。MDI ではメインウィンドウがメニューバーを持っており、個々のドキュメントはその内部ウィンドウに表示される。タイトルはあるがメニューバーは持っていない。メインウィンドウに含まれる内部ウィンドウは、ユーザーがサイズを変更したり最小化させたりできる。さらに、ウィンドウを重ねて表示したり並べて表示したりすることもできる。

[5] http://jcp.org/en/jsr/detail?id=296
[6] http://kenai.com/projects/guts
[7] http://horstmann.com/gridworld

開発者の多くは MDI を嫌っており、この形式のユーザーインターフェイスは時代遅れになってしまった。しばらくすると、シングルドキュメントインターフェイス (SDI) つまり一つのアプリケーションが複数のトップレベルウィンドウを表示する形式のほうが優れていると見なされるようになった。これはおそらく、SDI のウィンドウならホスト OS の標準的なウィンドウ管理ツールで管理できるようになるからだろう。トップレベルウィンドウが多すぎるのも考え物だということがはっきりしてきた頃に登場したのがタブインターフェイスだ。ここにきて再び、複数のドキュメントを一つのウィンドウにまとめるようになった。しかし今度はそのすべてをフルサイズで表示して、タブで選択できるようにする方式だ。二つのドキュメントを並べて比較することはできなくなったが、最終的にこの方式が勝ち残ったようだ。

Violet の最初のバージョンは MDI 形式だった。Java API には内部ウィンドウの機能があったが、ウィンドウを並べて表示したり重ねて表示したりする機能は自分で追加する必要があった。そして、後に Alexandre がタブインターフェイスに変更した。こちらのインターフェイスは Java API できちんとサポートされていた。ドキュメントの表示ポリシーが開発者から透過的に扱え、そして利用者からも選択可能なアプリケーションフレームワークが求められる。

Alexandre はさらに、サイドバーやステータスバー、ようこそ画面、スプラッシュスクリーンなどにも対応した。これらはすべて、本来なら Swing アプリケーションフレームワークに含まれているべきものだ。

22.8 Undo/Redo

複数回の undo/redo を実装するのは手強い作業に思えるが、Swing undo パッケージ ([Top00], Chapter 9) がよい設計指針になる。`UndoManager` が `UndoableEdit` オブジェクトのスタックを管理する。それぞれのオブジェクトが undo メソッドを持ち、このメソッドが編集操作を取り消す。そして `redo` メソッドは、取り消し自体を取り消す (つまり、本来の編集操作をもう一度実行する)。`CompoundEdit` は一連の `UndoableEdit` 操作で、undo や redo は全体をひとまとめにして行わないといけない。小規模でアトミックな編集操作 (グラフの場合だと、単一のエッジやノードの追加・削除など) を定義し、それを必要に応じて複合編集にまとめるとよい。

小規模でアトミック、そして簡単に undo できる操作を定義するのは難しい。Violet ではこんな感じになる。

- ノードやエッジの追加あるいは削除
- 子ノードのアタッチあるいはデタッチ
- ノードの移動
- ノードあるいはエッジのプロパティの変更

これらの操作はどれも、undo の方法がはっきりしている。たとえばノードの追加を undo したければ、そのノードを削除すればよい。ノードの移動を undo したければ、逆向きのベクターを使って移動させればよい。

図 22.6: Undo 操作はモデル内の構造の変更を取り消す必要がある

注意すべきなのは、こういったアトミックな操作が、ユーザーインターフェイス上での操作やその元となる Graph インターフェイスのメソッドと必ずしも一致するとは**限らない**ということだ。たとえば図 22.6 のようなシーケンス図で、ユーザーがアクティベーションバーをドラッグして右側のライフラインに移動させたとしよう。マウスのボタンを離したときには、こんなメソッドが実行される。

```
public boolean addEdgeAtPoints(Edge e, Point2D p1, Point2D p2)
```

このメソッドはエッジを追加するだけでなく、それ以外の操作も行う。その操作にかかわる Edge や Node のサブクラスで指定されているものだ。今回の場合だと、アクティベーションバーが右側のライフラインに追加されることになる。この操作を undo するには、アクティベーションバーも削除しなければいけない。つまり、**モデル** (今回の場合はグラフ) 自身がその構造の変更を記録して、あとで取り消せるようにしておく必要がある。コントローラの操作を記録するだけでは不十分だ。

Swing undo パッケージが示すように、グラフやノードそしてエッジのクラスは、もし構造の変更があったら UndoableEditEvent 通知を UndoManager に送らないといけない。Violet はもう少し汎用的な設計を採用しており、グラフ自身がこんなインターフェイスのリスナーを管理している。

```
public interface GraphModificationListener
{
  void nodeAdded(Graph g, Node n);
```

```
  void nodeRemoved(Graph g, Node n);
  void nodeMoved(Graph g, Node n, double dx, double dy);
  void childAttached(Graph g, int index, Node p, Node c);
  void childDetached(Graph g, int index, Node p, Node c);
  void edgeAdded(Graph g, Edge e);
  void edgeRemoved(Graph g, Edge e);
  void propertyChangedOnNodeOrEdge(Graph g, PropertyChangeEvent event);
}
```

フレームワークが各グラフにリスナーをインストールし、それが undo マネージャーとの橋渡しをする。undo をサポートするだけなら、モデルにこんな汎用リスナーを追加するというのはやり過ぎだ。グラフ上の操作を直接 undo マネージャーに通知すればいいのだから。しかし私は、将来的に共有編集機能を追加したいという希望もあったのだ。

もしあなたが自分のアプリケーションで undo/redo をサポートしたいのなら、そのアプリケーションのモデルにおいてアトミックな操作とは何かを伸長に考えるようにしよう (ユーザーインターフェイス上でアトミックかどうか、ではない)。モデルの構造が変わったときにはイベントを発生させ、Swing undo マネージャーがイベントをとりまとめられるようにすればいい。

22.9　プラグインアーキテクチャ

2D グラフィックになじみの深いプログラマーなら、新たな図形の型を Violet に追加するのもそんなに難しくない。たとえばアクティビティ図は、サードパーティから寄贈されたものだ。路線図や ER 図を描く必要が出てきたら、Visio や Dia をいじくり回すよりも Violet の拡張を書いたほうがよっぽど早い (どちらの型も、一日もあれば実装できる)。

型を実装するのには、Violet フレームワーク全体を把握している必要はない。グラフとノード、そしてエッジのインターフェイスを知っていさえすれば、お手軽に実装できるのだ。貢献者たちがフレームワーク自体の変化を気にせずに済むように、私はシンプルなプラグインアーキテクチャを設計した。

もちろんプラグインアーキテクチャを持つプログラムはいくらでもある。入念に作り込まれたアーキテクチャを持つものだ。Violet は OSGi に対応すべきだと誰かに言われたことがあったが、あのときは身震いしたね。代わりに、最小限の機能を持つ単純な仕組みを実装したんだ。

貢献者たちは、単にグラフやノードそしてエッジの実装を含む JAR ファイルを作ってそれを `plugins` ディレクトリに置くだけでいい。Violet が起動するときにこれらのプラグインが読み込まれる。このときに使うのは Java の `ServiceLoader` クラスだ。このクラスは、JDBC ドライバのようなサービスを読み込むために作られたものである。`ServiceLoader` は、指定したインターフェイス (今回の場合は `Graph` インターフェイス) を実装しているであろう JAR ファイルを読み込む。

個々の JAR ファイルはサブディレクトリ META-INF/services を持つ必要がある。この中には、インターフェイスの完全修飾名 (com.horstmann.violet.Graph など) と同じ名前のファイルがあり、その中にはすべての実装クラスの名前が 1 行ずつ記されている。ServiceLoader はプラグインディレクトリ用のクラスローダーを作り、すべてのプラグインを読み込む。

```
ServiceLoader<Graph> graphLoader = ServiceLoader.load(Graph.class, classLoader);
for (Graph g : graphLoader) // ServiceLoader<Graph> implements Iterable<Graph>
  registerGraph(g);
```

これは標準の Java が持つ単純ながらも便利な機能であり、あなた自身のプロジェクトでもうまく使えるかもしれない。

22.10 結論

オープンソースのプロジェクトではよくあることだが、Violet が産まれたきっかけは、必要な作業 (シンプルな UML をお手軽に作る) をこなせるものがなかったことだ。Violet がここまでに至ったのは、すばらしい Java SE プラットフォームによるところが大きい。このプラットフォームからはさまざまな技術を利用している。本章では、Violet が Java Beans や長期持続性、Java Web Start、Java 2D、Swing の Undo/Redo、サービスローダーなどの機能をどのように使っているかを解説した。これらの技術は、Java そのものや Swing に比べるとあまり知られていない。しかし、これらを活用すればデスクトップアプリケーションのアーキテクチャを劇的に単純化できる。これらのおかげで、空き時間に一人で作っていたプログラムがほんの数か月で立派なアプリケーションになったのだ。また、こういった標準の仕組みを使うことで、他の人たちが Violet を改良したりその一部を自分のプロジェクトに取り込んだりといったことも簡単にできるようになった。

第23章

VisTrails

Juliana Freire, David Koop, Emanuele Santos,
Carlos Scheidegger, Claudio Silva, and Huy T. Vo

VisTrails[1] は、データの調査や可視化をサポートするオープンソースのシステムである。科学ワークフローシステムやビジュアライゼーションシステムの便利な機能を含んでおり、今も成長を続けている。Kepler や Taverna といった科学ワークフローシステムと同様、VisTrails でも計算プロセスの指定ができ、既存のアプリケーションや疎結合のリソースそしてライブラリなどをルールに沿って統合できる。AVS や ParaView などのビジュアライゼーションシステムのように、VisTrails でも科学的な情報の可視化ができるようになっている。それを使ってデータを調べたり、さまざまな可視化表現でデータを比較したりできる。結果としてユーザーは複雑なワークフローを作れるようになる。これは科学的な発見のための重要なステップだ。データの収集から複雑な解析や可視化まで、すべてが一つのシステムに統合されているのだ。

VisTrails の特徴は、その履歴管理基盤だ [FSC+06]。VisTrails は、調査作業中に発生したデータや作業手順を記録してその履歴を管理する。繰り返すタスクを自動化するときに従来使われてきたのはワークフローだが、自然界の調査用のアプリケーション（データ解析や可視化など）では繰り返しはほとんど発生しない。変化するのが普通だ。ユーザーは、データに基づいた仮設をたててそれを検証する。それぞれ異なるけれども関連している一連のワークフローを作り、繰り返しながら調整していく。

VisTrails は、こういった常に成長し続けるワークフローを管理するために作られた。生成したデータ（可視化やプロットなど）、そこから派生するワークフロー、そしてその実行などの履歴を管理する。またアノテーション機能も提供しており、自動的にキャプチャした履歴を飾ることもできる。

結果を再生できるようにするだけでなく、VisTrails はその履歴情報を一連の操作として活用する。直感的なユーザーインターフェイスを使って、共同でデータを解析できるのだ。特筆すべきなのは、このシステムでは一時的な結果を保存しており、それを反映した推論にも

[1] http://www.vistrails.org

対応しているという点だ。これを利用すると、何かの結果に至った操作を吟味して前の推論や次の推論につなげられる。ユーザーはワークフローの各バージョンを直感的に移動でき、変更を取り消しても一切結果を失わずに済む。また、複数のワークフローをビジュアルに比較して、その結果をビジュアライゼーションスプレッドシートに並べて表示できる。

VisTrails は、ワークフローシステムやビジュアライゼーションシステムを広く取り入れるときの障害になる重大な使い勝手問題にも対応している。プログラミングの専門知識がない人も含めた幅広いユーザー層に対応するため、ワークフローの設計や利用を簡単にする一連の操作やユーザーインターフェイスを用意したのだ [FSC+06]。ワークフローの作成や修正にアナロジーを使えるようにしたり、実例でワークフローを問い合わせられるようにしたり、リコメンデーションシステムで対話的にワークフローを作っていけるようにしたりといったものだ [SVK+07]。また、新たなフレームワークも作った。これを使えば、プログラミングに詳しくないエンドユーザーでもカスタムアプリケーションを作って配布できるようになる。

VisTrails の拡張性の元になっているのはその基盤で、ユーザーにとってシンプルになるように作られている。ツールやライブラリを組み込んだり、お手軽に新機能のプロトタイプを作ったりできるようになっている。これは、このシステムをさまざまな分野に適用するのに役立っており、環境科学や精神医学、天文学、宇宙論、高エネルギー物理学、量子物理学、分子モデリングなどで使われ散る。

システムをオープンソースで自由に使えるようにするため、私たちは VisTrails を作るときに、自由なオープンソースのパッケージだけを使うようにした。VisTrails は Python で書かれており、GUI ツールキットには Qt を (その Python バインディングである PyQt と通して) 使っている。さまざまなユーザーがいるしさまざまなアプリケーションがあるので、システムを設計するときにはポータビリティを重視した。VisTrails は Windows でも Mac でも Linux でも動く。

23.1　システムの概要

データの調査は本質的にクリエイティブな作業であり、いろんなことをしなければならない。関連するデータを特定したり、そのデータを組み合わせて可視化したり、別の解法を追求する仲間と共同作業をしたり、自分の成果を広めたりといったことだ。科学調査の世界での一般的なデータ量や分析の複雑性を考慮すると、そのクリエイティビティをサポートするためには何らかのツールが必要となる。

こういったツールがうまく協調できるようにするためには次の二つの基本要件を満たさないといけない。まず、調査の手順をフォーマルな形式で指定できること。それが実行可能ならさらに望ましい。次に、その調査手順をそのまま再生したり、すこし変更して実行したりできること。作業の履歴を体系的に記録するための仕組みが必要だ。VisTrails は、こういった要件を念頭に置いて作られた。

図 23.1: VisTrails のユーザーインターフェイスのコンポーネント

ワークフローおよびワークフローベースのシステム

ワークフローシステムがサポートするのは、複数のツールを組み合わせたパイプライン (ワークフロー) づくりだ。繰り返し作業を自動化したり、作業を再生できるようにしたりする。さまざまなタスクに関するワークフローはすぐにプリミティブなシェルスクリプトに置き換えることができ、さまざまなワークフローベースのアプリケーションで確かめることができる。商用ツール (Apple の Mac OS X Automator や Yahoo! Pipes など) でも学術ツール (NiPype や Kepler そして Taverna など) でもかまわない。

ワークフローにはさまざまな利点があり、スクリプトや高級言語でのプログラムよりも優れている。シンプルなプログラミングモデルが用意されており、一連のタスクを作るのは単にあるタスクの出力と別のタスクの入力をつなぐだけのことだ。図 23.1 は、気象情報を含む CSV ファイルを読み込んで散布図を作るワークフローの例だ。

このシンプルなプログラミングモデルのおかげで直感的なビジュアルプログラミングインターフェイスを提供でき、**プログラミングの経験が浅いユーザーにもより使いやすい**ものとなった。また、ワークフローは**明確な構造**を持っている。グラフとして表すことができ、このとき各ノードはプロセス (あるいはモジュール) とそのパラメータを表す。またエッジはプロセス間のデータの流れを捕らえたものとなる。図 23.1 の例で考えると、`CSVReader` モジュールがファイル名 (`/weather/temp_precip.dat`) をパラメータとして受け取ってそのファイルを読み、ファイルの中身を `GetTemperature` モジュールと `GetPrecipitation` モジュールに

渡すことになる。これらがそれぞれ気温と降水量の値を matplotlib の関数に送って、散布図を生成する。

ほとんどのワークフローシステムは、特定の領域に特化した作りになっている。たとえば Taverna のターゲットはバイオインフォマティクスのワークフローだし、NiPype は神経画像のワークフローを作れるようになっている。VisTrails は他のワークフローシステムが持つ機能の多くに対応しているが、より幅広い分野の汎用的な調査タスクに対応するよう作られている。複数のツールやライブラリ、サービスの統合にも対応している。

データおよびワークフローの履歴

結果 (や生成データ) の履歴情報を保持しておくことの重要性は、自然科学の世界ではよく知られている。生成データの履歴 (あるいは監査証跡や系統、血統などとも呼ばれる) には、プロセスに関する情報やデータを生成するために利用したデータなどの情報が含まれる。履歴に含まれる情報は、データを永続化させたりその品質や所有者を調べたり、結果を再現して検証したりするための重要な鍵となる [FKSS08]。

履歴の中で重要となる情報の一つが、**因果関係**だ。これは、プロセス (一連の手順) に関する説明と入力データおよびパラメータをまとめたものであり、生成データを作るもとになったものだ。従って、履歴の構造は、指定した結果セットを作るもとになったワークフロー (あるいはワークフロー群) の構造を反映したものとなる。

実際、自然科学の分野でワークフローシステムが幅広く使われるようになったきっかけは、履歴情報の取得を簡単に自動化できることだった。初期のワークフローシステムは、履歴情報を取り込むように**拡張して**使っていた。一方 VisTrails は、**はじめから**履歴の取得に対応するように作られている。

ユーザーインターフェイスおよび基本機能

システムで使われているさまざまなユーザーインターフェイスコンポーネントを図 23.1 と図 23.2 にまとめた。ユーザーは、ワークフローを作ったり編集したりするときにワークフローエディタを使う。ワークフローグラフを作るには、モジュールレジストリからモジュールをドラッグして、それをワークフローエディタのキャンバスにドロップする。VisTrails には組み込みのモジュールが用意されており、それ以外にユーザーが自分でモジュールを追加することもできる (詳細は 23.3 節を参照)。モジュールを選ぶと、VisTrails はそのパラメータをパラメータ編集領域に表示する。ユーザーは、その値を設定したり変更したりできる。

ワークフローを修正すると、システムがその変更を捕捉して、後述するバージョンツリービューで確認できるようになる。また、Vistrails Spreadsheet を使えば、ワークフローやその結果を対話的に操作することもできる。スプレッドシートの各セルが、ワークフローのインスタンスを表す。図 23.1 では、ワークフローエディタの例で示したワークフローがスプレッ

図 23.2: 探索の履歴のアノテーションによる拡張

ドシートの左上のセルに表示されている。ワークフローのパラメータを直接変更することもできるし、そのパラメータをスプレッドシート内の別のセルと同期させることもできる。

バージョンツリービューを使えば、一つのワークフローのさまざまなバージョンを渡り歩ける。図 23.2 に示すように、バージョンツリーのノードをクリックすれば、ワークフローやそれに関連する結果 (ビジュアライゼーションプレビュー) およびメタデータを確認できる。メタデータの中には自動的に記録されるものもある (ワークフローを作ったユーザーの id や作成日など) が、自分でメタデータを追加することもできる。たとえば、ワークフローを識別するためのタグやワークフローについての説明などを追加できる。

23.2 プロジェクトの歴史

VisTrails の最初のバージョンは、Java と C++で書かれていた [BCC+05]。C++版はごく一部のアーリーアダプターに向けて公開したのだが、システムの要件を固めるうえで、そのフィードバックはとても有用だった。

Python ベースのライブラリやツールがさまざまな科学技術コミュニティで使われるようになってきたので、VisTrais の基盤もその後 Python に移行することにした。Python は、科学技術ソフトウェアでのモダンなグルー言語としての地位を急速に固めていった。Fortran や C、C++などで書かれたさまざまなライブラリが、Python バインディングを使ってスクリプト機能を提供するようになったのだ。VisTrails の狙いはワークフローで使うさまざまなソフトウェアライブラリの協調を促進することだったので、Python で実装してしまったほうが作業がずっ

図 23.3: VisTrails のアーキテクチャ

と楽になった。特に役立ったのが、Python の動的コードローディング機能だ。これはかつて LISP 環境にあったのと同じものだが、Python の開発者コミュニティのほうがずっと規模が大きいし、Python は標準ライブラリが非常に充実している。2005 年後半に、Python/PyQt/Qt を使った現在の版の開発が始まった。この選択のおかげで、システムの拡張がとてもシンプルにできるようになった。特に、新たなモジュールやパッケージを追加するのがシンプルになった。

VisTrails システムのベータ版が最初に公開されたのは 2007 年 1 月だ。その後、かれこれ 25,000 回以上はダウンロードされている。

23.3 インサイド VisTrails

ここまでで取り上げてユーザーインターフェイスの機能をサポートする内部コンポーネントは、VisTrails の上位アーキテクチャ内で図 23.3 のような構成になっている。Workflow の実行を制御するのが実行エンジンで、実行された操作やそのパラメータを追跡したりワークフローの実行履歴を記録したりする。実行中に、VisTrails では中間結果をメモリやディスクにキャッシュすることもできる。23.3 節で説明するように、モジュールとパラメータの組み

合わせがこれまでとは異なる場合にだけ再実行をする。このときの実行は、ベースになるライブラリ (matplotlib など) の適切な関数を起動する。ワークフローの結果、およびそれに関連づけられた履歴は、電子文書にもできる (23.4 節)。

ワークフローへの変更に関する情報を捕捉するのはバージョンツリーで、これはさまざまなストレージバックエンドを使って永続化できる。ローカルディレクトリの XML ファイルに保存してもかまわないし、リレーショナルデータベースを使ってもかまわない。VisTrails にはクエリエンジンも搭載されており、ユーザーが履歴情報を問い合わせることもできる。

特筆すべきことがある。VisTrails はもともと対話的に操作するツールとして作られたものだが、サーバーモードでも使えるということだ。ワークフローをいったん作ったら、それを VisTrails サーバーで実行できる。この機能は、さまざまな場面で便利に使える。たとえば Web ベースのインターフェイスを作ってユーザーにワークフローを操作させることもできるし、ワークフローをハイパフォーマンスコンピューティング環境で実行することもできる。

バージョンツリー: 変更ベースの履歴

図 23.4: 変更ベースの履歴モデル

VisTrails で導入した新たな概念の一つが、ワークフローの変化の履歴だ [FSC+06]。これまでのワークフローシステムやワークフローベースの可視化システムはどれも、生成されたデータの履歴しか管理できなかった。VisTrails はワークフローをファーストクラスのデータ項目として扱い、その履歴を保持する。ワークフローの変化の履歴を管理できるようになったことで、内省的な推論をしやすくなった。ユーザーは複数の推論を同時に進めることがで

き、その際に一切結果を失うこともない。また、中間の結果もシステムが保持しているので、その情報を使った推定もできる。一連の操作をまとめられるようにもなっており、調査プロセスを単純化できる。たとえば、何かのタスク用に作られたワークフローの中をあちこち移動しながらワークフローやその結果をビジュアルに比較したり (図 23.4 を参照)、大規模なパラメータ空間を調べたりといったことが簡単にできるようになる。さらに、履歴情報を問い合わせてその結果から学ぶこともできる。

ワークフローの変化を記録するために使うのが、変更ベースの履歴モデルだ。図 23.4 に示すとおり、VisTrails はワークフローへの操作や変更 (モジュールの追加やパラメータの変更など) をデータベースのトランザクションログと同様に扱う。この情報は木構造で表され、各ノードがワークフローのバージョンに対応する。そして、親ノードと子ノードの間のエッジが、親のワークフローから子のワークフローを得るために適用した変更を表す。私たちは、この木構造を表す用語としてバージョンツリーと vistrail(*visual trail* を短縮したもの) を同じ意味で使っている。変更ベースのモデルは、パラメータの値への変更とワークフロー定義への変更を同じ形式で扱うことに注意しよう。この一連の変更の情報があれば、生成データの履歴を得るのに十分だし、ワークフローがどのように変化してきたかの情報も記録できる。このモデルはシンプルかつコンパクトにできている。ワークフローの複数の**バージョン**を保持するシステムは他にもあるが、それらに比べて必要な容量は大幅に少なくなった。

この方式を採用したことによる利点は数多い。図 23.4 は、VisTrails で二つのワークフローを比較するときのビジュアル差分機能を示したものだ。ワークフローそのものがグラフで表されていたとしても、変更ベースのモデルを使えば二つのワークフローは極めて単純に比較できる。バージョンツリーをたどって、あるワークフローから別のワークフローへの変換に要する操作群を特定すれば、それで十分だ。

さらにもう一つ、重要な利点がある。ベースとなるバージョンツリーを使って、複数での共同作業ができるという点だ。ワークフローの設計はとても難しい作業だと言われており、複数名での共同作業が必要になることが多い。バージョンツリーのおかげで、別のユーザーの作業内容を直感的に可視化できる (対応するワークフローを作ったユーザーのノードだけに色を付けるなど) だけでなく、モデルの単調性のおかげで、複数のユーザーの変更を同期させるアルゴリズムも単純にできる。

ワークフローを実行したら、履歴の情報は簡単に記録できる。いったん実行が完了したら、生成データとその履歴との**強い**つながりを維持することも大切になる。つまり、そのデータを生成するために使ったワークフローとパラメータそして入力ファイルの情報だ。データファイルや履歴情報を移動したり変更したりしたら、その履歴に関連するデータを探したり、データに関連する履歴を探したりするのが難しくなることもある。VisTrails では永続ストレージの仕組みを提供しており、入力データや中間データそして出力データのファイルを管理できる。これで、履歴とデータの関連を強化しているのだ。この仕組みのおかげで、データの再生可能性がより高まった。履歴情報が参照するデータをすぐに (そして正確に) 特定できることが保証されるからだ。この方式で管理する大きな利点はもう一つある。中間データをキャッシュして他のユーザーと共有できることだ。

ワークフローの実行とキャッシュ

VisTrails の実行エンジンは、新しいツールやライブラリを既存のものと統合できるように作られた。サードパーティの視覚化ソフトや科学計算ソフトをラップするさまざまな手法に対応するよう試みた。特に、VisTrails はコンパイル済みのバイナリ (シェル上で実行したり入出力ファイルとして使うもの) と C++/Java/Python のクラスライブラリ (入出力の内部オブジェクトとして使うもの) の両方を統合できるようになっている。

VisTrails が採用しているデータフロー実行モデルでは、各モジュールが計算を行って、その結果生成されるデータがモジュール間の接続を通じて流れる。モジュールの実行はボトムアップ形式になり、各入力は、その場で上位モジュールを再帰的に実行して得る (モジュール A がモジュール B の**上位モジュール**であるとは、A から B へとつながる接続の流れがあることを意味する)。中間データは一時的に、(Python オブジェクトとして) メモリに格納されるか、あるいは (データにアクセスするための情報を含む Python オブジェクトでラップして) ディスクに格納される。

ユーザーが自分で VisTrails に機能を追加できるように、拡張可能なパッケージシステムも作った (23.3 節を参照)。パッケージを使えば、自作のモジュールやサードパーティのモジュールを VisTrails のワークフローに組み込める。パッケージの開発者は、計算用のモジュール群を特定しないといけない。そして、個々のモジュールについて、入出力のポートと計算の定義を示す。既存のライブラリの場合は、入力ポートからの入力を既存の関数へのパラメータに変換したり、結果の値を出力ポート用にマップしたりする計算メソッドが必要になる。

調査作業では、同じ構造を共有するよく似たワークフローを続けて実行することがよくある。ワークフローの実行効率を上げるために、VisTrails は中間結果をキャッシュして再計算の量を最小限に抑える。前の実行結果を再利用するので、キャッシュできるモジュールが機能することが前提となる。つまり、同じ入力を与えたときには同じ出力を生成するようになっているということだ。この要件のせいで、クラスの振る舞いには大きな制約が課されることになる。しかし私たちは、その制約は妥当なものであると考えている。

しかし、どうがんばってもこの挙動を実現できない場面もある。たとえば、リモートサーバーにファイルをアップロードするモジュールやファイルをディスクに保存するモジュールは、大きな副作用を持つモジュールである割にはその出力はあまり重要ではない。あるいは、乱数を使うモジュールがあって、結果が予測できないことが要件にあったとしよう。そんなモジュールについては、キャッシュ不能だというフラグを立てることができる。しかし、自然に機能させることができないものでもキャッシュできるよう変換することもできる。たとえば、データを二つのファイルに書き出す関数をラップして、ファイルの内容を出力するように変換するといったものだ。

データのシリアライズと格納

履歴をサポートするあらゆるシステムでポイントとなるのが、データのシリアライズと格納だ。VisTrails はもともと、データを XML 形式で格納していた。シンプルに、fromXML メソッドと toXML メソッドを内部オブジェクト (バージョンツリーや各モジュールなど) に組み込んでいたのだ。これらのオブジェクトのスキーマ変更に対応するために、各メソッドはスキーマのバージョン間での変換もコード化していた。プロジェクトが成長してユーザー数が増えたのに伴い、それ以外のシリアライズ方式にも対応することを決めた。リレーショナルデータベースなどにも対応しようとしたのだ。さらに、スキーマオブジェクトが成長するにつれて、共通のデータ管理基盤をよりよいものにする必要に迫られた。スキーマのバージョン管理やバージョン間での変換、そしてエンティティのリレーションシップのサポートなどだ。そのため、私たちは新たにデータベース (DB) レイヤーを追加した。

DB レイヤーは三つのコアコンポーネントで構成されている。ドメインオブジェクトとサービスロジック、そして永続化メソッドだ。ドメインと永続化のコンポーネントはバージョン管理されており、スキーマのバージョンごとに、どのバージョンのクラス群を使うかが決まる。このようにして、各バージョンのスキーマがどのコードを読むのかを管理している。また、あるバージョンのスキーマから別のバージョンのスキーマに変換するためのクラスも定義している。サービスクラスが提供するメソッドは、データとの間のインターフェイスとなるものであったりスキーマのバージョンの検出や変換を行うものであったりする。

このコードの大半は、つまらないコードの繰り返しになる。そこで、テンプレートとメタスキーマを用いて、オブジェクトの配置 (およびメモリ内でのインデックス) とシリアライズのコードを定義することにした。メタスキーマは XML で書かれており、デフォルトの XML 変更してシリアライズを拡張したり、VisTrails が定義するリレーショナルマッピングを追加したりできる。これは Hibernate[2] や SQLObject[3] といったオブジェクトリレーショナルマッピングフレームワークと似ている。しかし、そこにさらに特別なルーチンを追加して、識別子の再マッピングやスキーマのバージョン間でのオブジェクトの変換などのタスクを自動化した。さらに、同じメタスキーマからいろんな言語用のシリアライズコードを生成できるようにした。もともとのコードはメタ Python で書かれていた。つまり、ドメインと永続化のコードを生成するときには、Python のコードとメタスキーマの変数を使っていた。しかし、今では Mako テンプレート [4] に移行している。

自動変換は、新しいバージョンのシステム用へのデータ移行が必要なユーザーにとって大切な機能だ。そこで、変換用のフックを用意して、開発者の苦労を減らすように設計した。各バージョン用のコードのコピーを管理しているので、変換用のコードは単に、あるバージョンと別のバージョンをマップするだけのことになる。ルートレベルでは、マップを定義してどのバージョンからどのバージョンに変換できるのかを設定する。各バージョンから見ればこれは、複数の中間バージョンのチェーンになるのが一般的だ。当初持っていたマップは正

---

[2] http://www.hibernate.org
[3] http://www.sqlobject.org
[4] http://www.makotemplates.org

方向のものだけだった。つまり、新しいバージョンから古いバージョンへの変換はできなかったということだ。しかし最近の版では、逆方向へのマッピングも追加された。

各オブジェクトには update_version メソッドがあり、これは、さまざまなバージョンのオブジェクトを受け取って現在のバージョンのオブジェクトを返す。デフォルトでは再帰的な変換を行い、古いオブジェクトの各フィールドを新しいバージョンのものにマップする。このマッピングのデフォルトは、同じ名前のフィールドにコピーするというものだ。しかし、フィールドごとに、このデフォルトの挙動を「オーバーライド」できる。オーバーライドとは、古いオブジェクトを受け取って新しいバージョンを返すメソッドのことだ。スキーマの変更の大半はごく一部のフィールドにしか影響しないので、デフォルトのマッピングでたいていの場合は対応できる。しかし、このオーバーライド機能のおかげで、ローカルの変更を柔軟に定義できるようになった。

パッケージや Python による拡張性

VisTrails の最初のプロトタイプは、あらかじめ用意したモジュール群だけが付属していた。VisTrails のバージョンツリーや複数の実行結果のキャッシュなどのアイデアを試すにはよい環境だったが、長期的な視点での実用性には大きな制約があった。

私たちは VisTrails を、計算科学用の基盤だととらえている。つまり、文字通り、他のシステムやプロセスを使っていく上での足場となる機能を用意すべきだということだ。そう考えたときに本質的に必要になるのが、システムの拡張性である。拡張性を持たせるための一般的な方法は、何らかの言語を用意して適切なインタプリタを書くことだ。実行時にきめ細やかな制御ができるので、この方法は魅力的だ。キャッシュに関する要件を考えると、この魅力はさらに増す。しかし、本格的なプログラミング言語を実装するというのは多大な努力を要する作業だし、そもそもの目標はそんなことではなかった。さらに重要なのは、ちょっと VisTrails を試してみたいという人がまったく新しい言語の学習を強いられるというのは問題外だということだ。

私たちが欲しかったシステムは、ユーザーが独自の機能を簡単に追加できるようなものだった。また、ソフトウェアの複雑なパーツもうまく表現できるような強力な機能も必要だった。一例として、VisTrails では VTK ビジュアライゼーションライブラリ [5] に対応している。VTK は約 1000 個のクラス群から構成されており、コンパイル時の設定や OS によってその数は変わる。それぞれの場合について個別にコードを書くというのは生産的が悪いし、まったくもって非現実的だった。必要なのは、何らかのパッケージが提供する VisTrails モジュール群を動的に定義できる機能だった。自然な流れで、VTK を複雑なパッケージのモデルとして考えるようになった。

私たちが最初にターゲットとしていた分野の一つが計算科学で、システムを設計した当時は Python が「グルーコード」としての地位を確立しつつあるころだった。ユーザー定義の VisTrails モジュールの挙動を Python で指定できるようにすれば、利用するためのハードルを

[5] http://www.vtk.org

下げられるだろう。実際やってみると、Python は動的定義クラス群やリフレクション用のすばらしい基盤になってくれた。Python のほぼすべての定義は、ファーストクラスの式として同じように扱える。Python のリフレクションにあるこの二つの機能が、パッケージシステムで重要な役割を果たしている。

- Python のクラスは type callable を呼んで動的に定義できる。その戻り値がクラスを表すものとなり、一般的な方法で定義した Python のクラスとまったく同じように使える。
- Python のモジュールは`__import__`関数を呼んで簡単にインポートでき、その戻り値は、標準の `import` 文で指定したものとまったく同じように振る舞う。モジュールのあるパスも、実行時に定義できる。

もちろん、Python を使えばいいことずくめというわけではない。まず、Python が持つ動的な性質のせいで、たとえば VisTrails のパッケージの型安全性を保証したくても、それは一般的に無理な話になる。さらに重要なのが、VisTrails モジュールの要件の中には Python では実現できないものがあるということだ。後述する参照透過性に関する要件などがそれにあたる。それでも、Python の文化的にそういった制約が課されるのは仕方がないと考える。そんな問題はあるが、Python はソフトウェアの拡張性を実現する上でとても魅力的な言語だ。

VisTrails のパッケージおよびバンドル

VisTrails パッケージは、モジュール群にカプセル化されている。ディスク上では、Python パッケージと同じような状態だ (しかしこれ、まぎらわしい名前だな…)。Python パッケージは Python ファイル群で構成されており、そのファイルの中で Python の関数やクラスなどが定義されている。VisTrails パッケージは、特定のインターフェイスを備えた Python パッケージとなる。つまり、特定の関数や変数を定義したファイルを含んでいる。最も単純な状態の VisTrails パッケージは、一つのディレクトリ内に二つのファイル`__init__.py` と `init.py` を含むものだ。

最初のファイル`__init__.py` は Python パッケージに必須のファイルで、いくつかの定義を書かないといけない。内容は固定だ。検出する手立てはないものの、これに従わない VisTrails パッケージがあるとすればそれはバグだろう。このファイルで定義する値は、そのパッケージに関するグローバルに一意な識別子 (ワークフローをシリアライズするときにモジュールを区別するために使うもの) とパッケージのバージョンだ (パッケージのバージョンは、ワークフローやパッケージをアップグレードするときに重要となる。23.4 節を参照)。このファイルには、関数 `package_dependencies` と `package_requirements` を含めることもできる。VisTrails モジュールは、ルートクラスである `Module` 以外の別の VisTrails モジュールのサブクラスとしても作れるようにしている。そのため、ある VisTrails パッケージが別のパッケージの振る舞いを継承していることもあり、あるパッケージの初期化をしてから別のパッケージを初期化しないといけないなどということも考えられる。こういった、パッケージ間の依存関係を指定するのが `package_dependencies` だ。一方 `package_requirements` 関数は、システムレ

ベルでのライブラリの要件を指定する。これがあれば、Vistrails はバンドルを使って自動的に要件を満たすようにできる。

バンドルとはシステムレベルのパッケージで、VisTrails がシステム固有のツールを管理するためのものである。RedHat の RPM や Ubuntu の APT のようなものだ。これらのプロパティがあれば、VisTrails が Python モジュールを直接インポートして適切な変数にアクセスするだけでパッケージのプロパティを定義できる。

もう一つのファイルである `init.py` には、実際の Vistrails モジュール定義のエントリポイントが含まれる。このファイルの最も重要な機能は、二つの関数 `initialize` と `finalize` の定義だ。`initialize` 関数は、すべての依存パッケージを有効にした後でそのパッケージ自身を有効にするときに呼ばれる。パッケージ内のすべてのモジュールの準備を、この関数で行う。もう一方の `finalize` 関数は、通常はランタイムリソースの解放に使う (たとえば、そのパッケージが作った一時ファイルの削除などができる)。

各 VisTrails モジュールは、パッケージ内で一つの Python クラスとして表される。このクラスを VisTrails に登録するには、パッケージ開発者が `add_module` 関数をモジュールごとに呼ぶことになる。VisTrails モジュールは Python クラスであればどんなものでもかまわないが、いくつかの要件を満たさないといけない。その一つは、VisTrails が定義する基本 Python クラスのサブクラスでないといけないということである。基本クラスの名前は、何のひねりもない `Module` だ。VisTrails モジュールでは多重継承も使える。しかし、そのクラスの中の一つだけが VisTrails モジュールにならないといけない。VisTrails モジュールツリーの中では、菱形継承は使えない。多重継承は、クラスの mix-in を定義するときに特に有用だ。親クラスで定義されたシンプルな振る舞いを組み合わせて、より込み入った振る舞いを作ることができる。

利用できるポートが VisTrails モジュールのインターフェイスを決める。そしてそれは、モジュールの見た目だけでなく他のモジュールとの接続性にも影響する。そのため、どのポートを使うのかは明示的に VisTrails 基盤に示さないといけない。その方法は、`initialize` のときに `add_input_port` と `add_output_port` を適切に呼び出すか、あるいはクラスごとのリスト `_input_ports` と `_output_ports` を VisTrails ごとに指定する。

各モジュールで行う計算は、`compute` メソッドをオーバーライドして指定する。モジュール間でのデータの受け渡しにはポート経由で行い、`get_input_from_port` メソッドと `set_result` メソッドを利用する。昔ながらのデータフロー環境の場合、実行順はリクエストがやってきた順で決まっていた。私たちの場合は、実行順を決めるのにワークフローモジュールをトポロジカル整列した。キャッシングアルゴリズムを機能させるためには非巡回グラフが必要なので、実行スケジュールはトポロジカル整列の逆順にした。これで、関数を呼ぶときに上位モジュールを実行してしまうことがなくなる。あえてこの方法を選んだ。これによって、各モジュールの振る舞いを他とは切り離して考えやすくなり、キャッシュ戦略もよりシンプルかつ堅牢にできた。

全体的な指針として、VisTrails モジュールでは、`compute` メソッドの評価中に副作用のある関数を使わないようにしている。23.3 節で説明したとおり、この要件のおかげでワークフ

ローの一部だけをキャッシュできるようになった。各モジュールがこの性質を尊重すれば、その振る舞いは上位モジュールの出力となる。下位にあるすべての非巡回グラフは単に一度ずつ計算するだけでよく、その結果は再利用できる。

データをモジュールとして渡す

VisTrails モジュールとその通信に独特な特徴の一つが、モジュール間で受け渡しされるデータ自体もまた VisTrails モジュールだという点だ。VisTrails では、モジュールクラスとデータクラスが一つの階層にまとまっている。たとえば、モジュールは、**自分自身**を計算結果として出力できる (そして実際のところ、すべてのモジュールはデフォルトで、出力ポート"self"を提供している)。この方式の主なデメリットは計算とデータを概念的に切り離せなくなることだ (データフローベースのアーキテクチャなら、これらはきちんと分離できる)。しかし、二つの大きなメリットがある。まず、このようにすると、Java や C++のオブジェクトの型システムに限りなく近づけることができる。これは決して偶然の産物ではなく、狙った結果だ。VTK のような巨大なクラスライブラリを自動でラップするためには、この機能が非常に重要だったのだ。これらのライブラリでは、オブジェクトの計算結果として別のオブジェクトを作れるようになっており、計算とデータを分離する方式でこれをラップしようとすると、とても複雑になってしまう。

もう一つのメリットは、ワークフローの中で定数を設定したりユーザーが変更できるパラメータを設定したりするのが簡単になるということだ。システムの他の部分に、違和感なく組み込めるようになる。こんな例を考えてみよう。あるワークフローで、定数で指定した Web 上の場所からファイルを読み込むとする。今はこの定数を GUI で指定している。URL はパラメータとして指定できる (図 23.1 のパラメータ編集領域を参照)。このワークフローを変更する自然な方法は、この仕組みを使って、どこか別の上位モジュールで**計算した** URL を取得するようにすることだ。ワークフローのその他の部分への変更は、可能な限り抑えたい。モジュールが自分自身を出力できるという前提なら、単に適切な値の文字列をパラメータに対応するポートにつなぐだけでよくなる。定数を出力した結果を自分自身で評価するので、その振る舞いは値を定数として渡された場合とまったく同じになる。

定数の設計に関しては、他にも考慮すべきことがある。定数の型によって、値を設定するために最適な GUI インターフェイスは異なる。たとえば、VisTrails の場合は、ファイル定数モジュールではファイル選択ダイアログを使う。Boolean 値はチェックボックスで指定するし、色の値は OS ネイティブの色選択ダイアログを使う。これを実現するには、開発者が `Constant` を継承したカスタム定数を作って、適切な GUI ウィジェットの定義と定数の文字列表現 (定数をディスクにシリアライズするときに使うもの) をオーバーライドしないといけない。

あと、シンプルなプロトタイピング用として、VisTrails では組み込みの `PythonSource` モジュールを提供している。このモジュールを使えば、スクリプトを直接ワークフローに流し

図 23.5: 新機能のプロトタイピングを PythonSource モジュールで行う

込める。PythonSource の設定ウィンドウ (図 23.5 を参照) から複数の入出力ポートを指定し、実行したい Python コードも指定する。

23.4 コンポーネントおよび機能

先述のとおり、VisTrails の提供する機能やユーザーインターフェイスを使うと、調査や計算のタスクを作ったり実行したりする作業を単純化できる。そのいくつかについて、ここで説明する。また、VisTrails を基盤として使って、履歴情報を含めて公開する方法についても簡単に説明する。VisTrails やその機能についてのより詳しい説明は、オンラインドキュメントを参照してほしい[6]。

Visual Spreadsheet

VisTrails で複数のワークフローをたどったり結果を比較したりするには、Visual Spreadsheet を使う (図 23.6 を参照)。このスプレッドシートは VisTrails のパッケージで、シートとセルからなる独自のインターフェイスを用意している。各シートにはセルのセットが含まれ、そのレイアウトもカスタマイズできる。セルの中身はワークフローが生成した結果をビジュアルに表現したもので、データの種類によって表示方法をカスタマイズできる。

スプレッドシート上にセルを表示するには、ワークフローに `SpreadsheetCell` の派生モジュールを含めないといけない。個々の `SpreadsheetCell` モジュールがスプレッドシート内の一つのセルに対応するので、一つのワークフローが複数のセルを生成することになる。`SpreadsheetCell` モジュールの `compute` メソッドが、実行エンジン (図 23.3) とスプレッド

[6] http://www.vistrails.org/usersguide

図 23.6: Visual Spreadsheet

シートの間の通信を担当する。実行中に、スプレッドシートはその型に合わせたセルをその場で作る。このときには、Python の動的クラス生成機能を活用する。したがって、独自のビジュアル表現を作りたければ、`SpreadsheetCell` のサブクラスを作って、その `compute` メソッドでスプレッドシートにカスタムセルタイプを送ればいい。たとえば、図 23.1 のワークフローにある `MplFigureCell` は、matplotlib が作った画像を表示するために用意した `SpreadsheetCell` モジュールだ。

スプレッドシートの GUI バックエンドとして PyQt を使っているので、カスタムセルウィジェットは PyQt の `QWidget` を継承したクラスにしないといけない。また、`updateContents` メソッドも定義する必要がある。このメソッドはスプレッドシートが実行するもので、新しいデータがやってきたときにそのウィジェットを更新する。各セルウィジェットでは、オプションでカスタムツールバーを用意することもできる。そのときには `toolbar` メソッドを実装する。このツールバーは、セルが選択されたときにスプレッドシートのツールバー領域に表示される。

図 23.6 は、VTK セルを選択したときのスプレッドシートの様子を示すものだ。このときのツールバーには、PDF にエクスポートしたりカメラの位置を保存してワークフローに戻ったりアニメーションを作ったりといった、専用のウィジェットが表示される。スプレッドシートパッケージではカスタマイズ可能な `QCellWidget` を用意しており、履歴の再生 (アニメー

ション) やマルチタッチのイベント転送などの共通機能を提供している。これを QWidget のかわりに使えば、新たなセルタイプを手っ取り早く開発できる。

スプレッドシートで使えるセルタイプは PyQt ウィジェットだけではあるが、他の GUI ツールキットで書かれたウィジェットを組み込むこともできる。そのためには、ウィジェットが自分の要素をネイティブプラットフォームに公開しないといけない。PyQt はそれを使ってウィジェットを操作する。私たちは VTKCell でこの手法を使った。実際のウィジェットが C++で書かれたものだったからだ。実行時に VTKCell が、ウィンドウ ID と (Win32、X11、Cocoa/Carbon などの) ハンドルを取得する。そしてそれをスプレッドシートのキャンバスにマップする。

セルと同様、シートもカスタマイズできる。デフォルトでは、各シートはタブビューの上にテーブルレイアウトで配置されている。しかし、スプレッドシートウィンドウからシートを切り離すこともでき、そうすれば複数のシートを同時に見ることができるようになる。また、新たなシートレイアウトを作ることもできる。そのためには StandardWidgetSheet のサブクラスを作ればいい。これもまた PyQt ウィジェットだ。StandardWidgetSheet は、セルのレイアウトだけを管理するのではない。編集モードでのスプレッドシートとセルのやりとりも管理する。編集モードでは、ユーザーがセルのレイアウトを操作して、複数のセルに対してより高度な操作を実行できる。単にセルの中身を扱う以上のことができるのだ。たとえば、類推を適用したり (23.4 節を参照)、パラメータを変えた新しいバージョンのワークフローを作ったりといった操作ができる。

Visual Differences および類推

VisTrails を作るときに私たちが考えたのは、履歴情報をただ取得するだけではなく、それを**活用**できるようにすることだった。まずは、バージョン間の正確な差分をユーザーが見られるようにしたかった。ただ、他のワークフローとの差分に関しては、もっと便利な機能も追加できることがわかってきた。これらが可能となったのは、VisTrails が履歴の変化を追跡していたからだ。

バージョンツリーがすべての変更を記録していていつでも元に戻せるので、あるバージョンから別のバージョンに変換するためのアクション群を探すことができる。中には他の変更をキャンセルするような変更もあるので、それを使ってアクション群をコンパクトにまとめることもできる。たとえば、あるモジュールを追加したけれども後にそれを削除したという場合は、差分を算出するときにそれらの変更を考慮しなくてもかまわない。最後に、ちょっとした経験則でアクション群を単純化する。両方のモジュールのそれぞれ別のアクションで同じモジュールが追加されている場合は、その追加と削除をキャンセルする。

変更のセットをビジュアルに表せば、モジュールや接続そしてパラメータなどの共通点と相違点を示せる。その様子を図 23.4 に示す。両方のワークフローに登場するモジュールや接続はグレーで表示され、どちらか一方にしか登場しないものは、そのワークフローに対応する色で表示される。同じモジュールに別のパラメータが渡されている場合は明るめのグレー

で表示され、そのモジュールのパラメータの違いを表で確認できる。この表には、それぞれのワークフローでのパラメータの値が表示される。

類推操作を使えば、これらの差分を別のワークフローに適用できるようになる。既存のワークフローに対する変更群 (精度の変更や出力画像のファイルフォーマットの変更など) があれば、それを別のワークフローにも適用できるのだ。そのためには、まず変換前と変換後のワークフローを選んで変更の範囲を絞り込み、さらにその類推を適用先のワークフローも指定する。VisTrails は最初の二つのワークフローの差分を算出してそれをテンプレートとし、その差分を三番目のワークフローにどのように適用するかを調べる。最初の状態が完全に一致していないワークフローであっても差分は適用できるので、ソフトなマッチングを使って同じモジュールの対応を判断できる必要があった。このマッチングを使えば差分をマップしなおせるので、選択したワークフローに変更群を適用できる [SVK+07]。この方法が必ずうまくいくというわけではなく、できあがるワークフローが望みのものとは違ってしまう可能性もある。そんな場合は、間違っている部分を修正したり、前のバージョンに戻って変更を手動で適用したりもできる。

類推の際にソフトマッチングを算出するには、ローカルマッチ (完全に一致するのか、ほぼ同じモジュールなのか) とワークフロー全体の構造とのバランスをとっておきたい。完全一致の算出は (部分グラフの写像が難しいから) 非効率的なので、経験則を使う必要がある。簡単に言うと、二つのワークフローでよく似たモジュールが同じ隣接モジュールを共有している場合は、その二つのモジュールの機能は同じであると判断し、一致すると見なすということだ。もう少し正確に言うと、二つのワークフローの積グラフを作る。各ノードは元のワークフロー内で考え得るモジュールのペアを表し、エッジは共有している接続を表す。そして各ノードからエッジ越しの隣接ノードへスコアを拡散する。これは Google の PageRank とよく似たマルコフ過程であり、最終的に収束したスコア群には何らかのグローバル情報が含まれている。このスコアを使って一番よいマッチングを判断し、閾値に満たないモジュールのペアを切り離す。

履歴の問い合わせ

VisTrails が記録する履歴に含まれる内容は、ワークフロー群の構造やメタデータそして実行ログだ。ユーザーがこれらのデータにアクセスして調べられるようにすることが重要になる。VisTrails は、テキストベースとビジュアル (WYSIWYG) なものとの二種類の問い合わせインターフェイスを用意している。タグやアノテーションそして日付といった情報について、オプションのマークアップでキーワード検索ができる。たとえば、キーワード `plot` を含むすべてのワークフローの中で `user:~dakoop` が作ったものを探すといった検索だ。しかし、あるワークフローの中の特定の部分グラフへの問い合わせについては、もっと簡単に表現できる。ビジュアルな、例示による問い合わせのインターフェイスを使ったもので、問い合わせをスクラッチから組み立てたり既存のパイプラインのパーツをコピーして修正したりできる。

この例示による問い合わせのインターフェイスを作るにあたり、大半のコードは既存のワークフローエディタのものを流用した。パラメータの構築あたりを多少変更しただけだ。パラメータに関しては、実際の値を指定するよりも値の範囲やキーワードで検索できたほうが便利なことが多い。そこで、パラメータの値フィールドを修正子を追加した。ユーザーがパラメータの値を追加したり変更したりしたときに、この修正子を選べるようになっている。デフォルトは、値をそのまま使うというものだ。ビジュアルに問い合わせを作るだけでなく、問い合わせの結果もビジュアルに表示される。マッチするバージョンがバージョンツリー内で強調表示され、選択したワークフローのなかでマッチする部分も強調される。別の問い合わせを開始したりリセットボタンをクリックしたりすると、問い合わせ結果のモードから抜けられる。

永続データ

VisTrails は、ある結果がどのような経緯で導かれたのかや、各ステップでどのように指定したのかんどといった履歴を保存する。しかし、ワークフローの実行の再現は、そのワークフローで使うデータが残っていないと難しい。さらに、実行に時間のかかるワークフローの場合は、中間データをキャッシュとして保存して複数セッション間で使い回せるようにしておけば、再計算の手間を省ける。

多くのワークフローシステムは、ファイルシステムでのデータへのパスを履歴に保存する。しかしこの方式には問題がある。ユーザーがファイル名を変更するかもしれないし、ワークフローを別のシステムに移動するときにデータをコピーしないかもしれない。あるいはデータの中身を変更してしまうかもしれない。いずれの場合にも、パスを履歴に含めていると不都合が起こる。データのハッシュを計算してそれを履歴として保存しておけば、データが変更されたかどうかを判断する助けにはなる。しかし、そんなデータがあったとして、それを見つける助けにはならない。この問題を解決するために、私たちは Persistence パッケージを作った。この VisTrails パッケージは、バージョン管理システムの基盤を使ってデータを格納し、履歴から参照できるようにする。現在は Git を使ってデータを管理しているが、他のシステムに対応するのも簡単だ。

データの特定には UUID(Universally Unique Identifier) を使い、git のコミットハッシュで特定のバージョンを参照する。あるワークフローの実行と別の実行でデータが変わったときには、新しいバージョンをリポジトリにチェックインする。つまり、`(uuid, version)` のタプルを複合キーとして、あらゆる状態のデータを取得できるようになる。データのハッシュを格納するだけでなく、さらにそのデータを生成した上位のワークフロー (入力ではない場合) のシグネチャも保存する。これで、別のデータへリンクできるようになり、同じ計算を再実行するときにもデータを再利用できるようになる。

このパッケージを作るときにまず考えたのは、ユーザーがデータを選んだり取得したりする方法をどうするかということだった。また、すべてのデータは同じリポジトリにまとめておきたかった。入力として使うものであろうが出力であろうが中間データ (あるワークフロー

の結果を別のワークフローへの入力として使う場合) であろうが同じにしたかったのだ。ユーザーがデータを特定する方法には大きく二つのモードがある。新しい参照を作るか既存の参照を使うかだ。最初の実行の後で、新しい参照は既存の参照となる。というのも、実行中に永続化されるからだ。ユーザーは、それを使って別の参照を作ることもできる。ただ、あまりそんなことはない。ユーザーは最新版のデータを使いたがることが多いので、バージョンを指定せずに参照を指したときは最新バージョンを指すことになっている。

モジュールを実行する前に、そのすべての入力を再帰的にアップデートしたことを思いだそう。永続データモジュールは、もし上流の計算が既に実行済みの場合は、入力をアップデートしない。アップデートするかしないかを判断するために、上流の部分ワークフローのシグネチャを永続リポジトリのものと比較し、もしそのシグネチャが存在すれば、計算済みの結果を取得する。さらに、データの ID とバージョンを履歴として記録し、その実行を再現できるようにする。

アップグレード

VisTrails の肝となる履歴機能を使えば、古いバージョンのワークフローをアップグレードして新しいバージョンのパッケージで動かせるようになる。パッケージはサードパーティが作ることもあるので、ワークフローをアップグレードするための仕組みだけではなくパッケージ開発者がアップグレード方法を指定するためのフックも必要になる。ワークフローのアップグレードで中心となる操作は、何かのモジュールを新しいバージョンで置き換えることだ。古いモジュールのすべての接続やそのパラメータを置き換えないといけないので、この操作は込み入ったものとなる。さらに、モジュールのインターフェイスが変わったりしたときには、パラメータや接続を再設定したり代入しなおしたり名前を変更したりしないといけなくなるかもしれない。

各パッケージ (およびそれに関連するモジュール) はバージョン番号のタグが付いており、バージョンが変わるということは、そのパッケージ内のモジュールも変わる可能性がある。中には変わらないものだってあるだろうし、場合によっては大半が変わらないままということだってある

でも、それを判断するにはコードを解析しないといけない。しかし私たちは、インターフェイスが変わっていないモジュールも自動的にアップグレードする方法にひかれた。そのために、まずはモジュールを新しいバージョンに置き換えて、うまく動かないときは例外を投げるようにした。開発者がモジュールのインターフェイスを変えたり名前を変えたりするときには、その変更を明示的に指定できるようにもした。これを管理しやすくするために、作ったのが `remap_module` メソッドだ。このメソッドを使えば、デフォルトのアップグレードで変更すべき箇所を開発者が定義できる。たとえば、入力のポート名を ‘file’ から ‘value’ に変更した場合は、その再マッピングを指定することができる。そうしておけば、新しいモジュールを作った時にも、古いバージョンの’file’ への接続がすべて’value’ につながるようになる。組み込みの VisTrails モジュールでのアップグレードパスの例を示す。

```
def handle_module_upgrade_request(controller, module_id, pipeline):
   module_remap = {'GetItemsFromDirectory':
                       [(None, '1.6', 'Directory',
                         {'dst_port_remap':
                              {'dir': 'value'},
                          'src_port_remap':
                              {'itemlist': 'itemList'},
                          })],
                   }
  return UpgradeWorkflowHandler.remap_module(controller, module_id, pipeline,
                                             module_remap)
```

このコード片は、古いバージョンの GetItemsFromDirectory モジュール (1.6 以前のもの) を使っているワークフローをアップグレードして、代わりに Directory モジュールを使わせるようにするものだ。旧モジュールの dir ポートを value にマップし、itemlist ポートを itemList にマップする。

アップグレードのたびに新たなバージョンツリーを作り、これでアップグレード前後の実行を区別したり比較したりできるようにする。アップグレードによってワークフローの実行結果が変わることもあり (パッケージ開発者がバグを修正した場合など)、その履歴情報を追いかけられるようにする必要がある。古いバージョンの vistrails では、ツリー内のすべてのバージョンをアップグレードしないといけないこともあった。無駄ながらくたを減らすため、最近はユーザーが選んだものだけをアップグレードするようにした。さらに、実際にワークフローを変更したり実行したりするときまでアップグレード結果を保存しないようにするという設定も選べるようにした。単に見るだけなら、アップグレードを保存する必要はない。

履歴を含めた結果の共有と公開

科学的方法は、再現できることが何よりも重要だ。その一方で、コンピュータによる実験結果を公開するときには、同じ実験を繰り返したり一般化したりするための十分な情報が提供されてこなかった。最近は、再現可能な結果を公開しようという動きが出ている。この方針に従おうとしたときの大きな問題は、結果を再現したり検証したりするのに必要な全コンポーネント (データ、コード、そしてパラメータ設定) をひとまとめにするのが難しいという事実だ。

詳細な履歴を記録する機能、そしてこれまでに紹介したさまざまな機能を活用することで、VisTrails はこのプロセスを単純化した。コンピュータによる実験の結果をシステムの外部に持ち出しやすくしたのだ。しかし、ドキュメントをリンクしたり、履歴情報を共有したりするための仕組みが必要になる。

そこで、そんな VisTrails パッケージを作った。結果を紙に出力するときに、その履歴情報にリンクできるようにするものだ。私たちが作った LaTeX パッケージを使えば、VisTrails のワークフローにリンクした図を組み込める。次の LaTeX コードは、ワークフローの結果を含む図を生成するものだ。

```
\begin{figure}[t]
\centering{
\vistrail[wfid=119,buildalways=false]{width=0.9\linewidth}
}
\caption{Visualizing a binary star system simulation. This is an image
  that was generated by embedding a workflow directly in the text.}
\label{fig:astrophysics}
\end{figure}
```

この文書を pdflatex でコンパイルすると、受け取ったパラメータを使って`\vistrail`コマンドが Python スクリプトを実行する。このスクリプトが XML-RPC メッセージを VisTrails サーバーに送って、`id 119`のワークフローを実行する。同じ Python スクリプトがワークフローの結果をサーバーからダウンロードし、LaTeX の`\includegraphics`コマンドにレイアウトオプション (`width=0.9\linewidth`) を指定したものを生成して結果を PDF 文書に組み込む。

VisTrails の実行結果は Web ページや Wiki に組み込むこともできるし、Word の文書や PowerPoint のプレゼンテーションにも組み込める。Microsoft PowerPoint と VisTrails とのリンクには、COM(Component Object Model) や OLE(Object Linking and Embedding) などのインターフェイスを使った。PowerPoint で使えるオブジェクトにするには、最低限の COM インターフェイス `IOleObject`、`IDataObject` そして `IPersistStorage` を実装しないといけない。私たちが OLE オブジェクトを作るために使ったのは Qt の `QAxAggregated` クラスで、これは COM インターフェイスの実装を抽象化したものなので、`IDataObject` や `IPersistStorage` は Qt が自動的に処理してくれる。したがって、唯一実装する必要があるのは `IOleObject` インターフェイスだけになる。このインターフェイスで一番重要な呼び出しが `DoVerb` だ。これを使って、VisTrails が PowerPoint からのアクション (オブジェクトのアクティブ化など) に対応できるようにする。VisTrails オブジェクトをアクティブにすると、VisTrails アプリケーションを読み込んで、ワークフローを開いたり操作したり挿入するパイプラインを選んだりできるようになる。VisTrails を閉じると、パイプラインの結果が PowerPoint に反映される。パイプラインの情報も、OLE オブジェクトとして保存される。

実行結果とそれに関連する履歴をユーザーが自由に共有できるようにするために作ったのが crowdLabs だ。[7] crowdLabs はソーシャルな Web サイトで、便利なツールやスケーラブルな基盤からなるものだ。科学者たちが共同作業で、データの分析や可視化をできるようにする環境を提供する。crowdLabs は VisTrails と密接に統合されている。VisTrails の実行結果を共有したければ、VisTrails から直接 crowdLabs サーバーに接続して情報をアップロードできる。情報のアップロードが完了すれば、Web ブラウザ上でワークフローを操作したり実行したりできるようになる。ワークフローの実行は、crowdLabs を動かしている VisTrails サーバーが行う。VisTrails を使って再実行可能な形式で公開する方法についての詳細は、`http://www.vistrails.org` を見てほしい。

[7] `http://www.crowdlabs.org`

23.5　教訓

データの調査と可視化ができて、履歴に対応したシステムを作ろうと考えだしたのは 2004 年だった。その当時はそれがいかに大変なことかも知らなかったし、今のような状態まで持ち込むのにどれくらいかかるかも見当がつかなかった。これが幸いした。もし当時の私たちがそれを知っていたら、そもそも作り始める気にもならなかっただろう。

最初の頃は、何かうまくいきそうな手が見つかったらすぐに新機能のプロトタイプを作って、それをユーザーに使ってもらっていた。ユーザーからのフィードバックや励ましのことばがプロジェクトの原動力だった。ユーザーからのフィードバックなしに VisTrails を作るのは不可能だっただろう。このプロジェクトの特徴として特筆すべき点は、システムの大半の機能がユーザーからの要望に応じて作られたものであるということだ。しかし、ユーザーからの要望の中には、彼ら/彼女らが本当にやりたいこととはずれているものも少なくなかった。ユーザーの要望に応えるというのは、単にユーザーの指示通りに作るってことじゃない。何度となく機能の設計を繰り返し、ほんとうに便利でうまくシステムに組み込めるようなものにしていった。

こんな風にユーザー主導の開発を進めているんだから、すべての機能が実際によく使われているのだろうと思うかもしれない。しかし残念ながら、そうでもない。その理由のひとつは、追加した機能の中にはあまりにも「独特な」機能、つまり他のツールにはまず見られないものもあったということだ。たとえば、類推機能は大半のユーザーにとってあまりなじみのないものだろうし、恐らくバージョンツリーだってそうだ。ユーザーがそういった機能に慣れるのには時間がかかる。さらにもうひとつ重要な理由は、ドキュメントが貧弱なことやそもそもドキュメントがないことだ。オープンソースのプロジェクトにはよくあることだが、新機能を開発しているほうが既存機能のドキュメントを書いているよりもずっと楽しかった。ドキュメントに問題があるせいで便利な機能がうまく活用されないこともあったし、それだけではなくメーリングリストへの質問も増加した。

VisTrails のようなシステムを使ううえで難しいのが、システムが汎用的なものであるという点だ。使い勝手を改良しようと最善の努力をしたところで、VisTrails は複雑なツールなので、使いこなすための学習曲線は急勾配になる。ドキュメントやシステムを改良したり、アプリケーションよりのサンプルや分野ごとのサンプル用意したりなどして、できるだけ学習の障壁を下げるようにしてきた。また、履歴の概念が一般に広まるにつれて、VisTrails の開発思想をユーザーがより理解しやすくなった。

謝辞

VisTrails に貢献してくれた優秀な開発者たちに感謝する。Erik Anderson、Louis Bavoil、Clifton Brooks、Jason Callahan、Steve Callahan、Lorena Carlo、Lauro Lins、Tommy Ellkvist、Phillip Mates、Daniel Rees、そして Nathan Smith だ。Antonio Baptista に感謝する。このプロジェクトのビジョンを確立する上で大きな助けになってくれた。さらに Matthias Troyer に

も感謝する。彼の助けのおかげでこのシステムはよりよいものとなり、特に履歴つきの公開機能の開発やリリースの大きな原動力になってくれた。VisTrails システムの研究や開発には National Science Foundation からの支援を受けた。IIS 1050422、IIS-0905385、IIS 0844572、ATM-0835821、IIS-0844546、IIS-0746500、CNS-0751152、IIS-0713637、OCE-0424602、IIS-0534628、CNS-0514485、IIS-0513692、CNS-0524096、CCF-0401498、OISE-0405402、CCF-0528201、CNS-0551724 だ。また、Department of Energy SciDAC (VACET and SDM centers) や IBM Faculty Awards からも支援を受けた。

第24章

VTK

Berk Geveci and Will Schroeder

Visualization Toolkit (VTK) は、データ処理や可視化用のソフトウェアシステムとして幅広く使われているものだ。科学計算や医用画像解析、計算幾何学、レンダリング、画像処理や情報科学などの分野で使われている。この章では、VTK の概要とその基本的な設計パターンに関して説明する。

何かのソフトウェアシステムについて完全に理解するには、単にそのソフトウェアが解決しようとしている問題が何かを理解するだけでは不十分だ。そのソフトウェアが生み出す文化も理解しないといけない。VTK に関して言うと、上っ面だけを見れば、科学データ用の 3D 可視化システムに過ぎない。しかし、それが生み出す文化を踏まえるとその背景には大きなストーリーがあり、なぜ開発されたのかやなぜこの形で公開されたのかなどを理解する助けになるだろう。

VTK の元になる概念が生まれて実際に書き始めた当時、作者 (Will Schroeder、Ken Martin、Bill Lorensen) はみな GE Corporate R&D の研究者だった。当時の私たちが使っていたのは、LYMB という先進的なシステムだった。これは Smalltalk 風の環境で、C 言語で実装されていた。当時としてはすばらしいシステムだったが、研究者としての私たちは二つの大きな障害に悩まされていた。知的所有権の問題と、非標準のプロプライエタリなソフトウェアを使っているという問題だ。知的所有権の問題のせいで、いったん顧問弁護士に口出しされるとソフトウェアを GE の外部に公開することは事実上不可能だった。さらに、ソフトウェアを GE の内部でだけ公開することにしたとしても、顧客の多くはプロプライエタリで非標準なシステムを覚えようとはしたがらないだろう。がんばってマスターしたところで、転職してしまえば何の役にも立たなくなる。さらに、標準のツールセットの恩恵を受けることもできない。そんな状況があったので、VTK を開発する最大の動機として「オープンな標準規格を使うこと」つまり「**コラボレーションプラットフォーム**として開発し、その技術を顧客にも使ってもらいやすくすること」がうまれた。VTK にオープンソースライセンスを適用したことは、これまでの設計上の判断の中でもおそらく最も重要なものだった。

最終的に、伝播性のないライセンスを選んだ (つまり、GPL ではなく BSD を選んだ) こと

は、今思えばとてもよい判断だった。そのおかげで、サービスやコンサルのビジネスもできるようになり、それが後の Kitware につながったのだから。当時の私たちが重視していたのは、学校や研究所そして企業などが共同作業をするための障壁を下げることだった。多くの組織では、伝播性のあるライセンスが避けられていた。問題に巻き込まれる恐れがあったからだ。実際私たちも、伝播性のあるライセンスのせいでオープンソースソフトウェアの普及がなかなか進まないと思っている。まあ、それはまた別の話。言いたいのは、ソフトウェアシステムを作るときにはライセンスの選択が重要のポイントの一つになるっていうことだ。プロジェクトのゴールを見定めて、知的所有権の問題に適切に対応することが大切だ。

24.1　VTK とは

VTK は、もともとは科学データの可視化システムだと考えられていた。外野の人間のほとんどは、可視化を単なる幾何学的なレンダリングに過ぎないと考えている。仮想的な物体を調べたり、それを使ったやりとりをしたりするというものだ。まあそれも可視化の一つの側面ではあるが、一般にデータの可視化といえば、データを感覚入力に変換する作業全体を意味する。感覚入力の典型例は画像だが、それだけではない。触覚や聴覚に対応する形式も含む。データの形式は、幾何学的かつ位相的な構造 (メッシュあるいは複雑な空間分解による抽象化など) だけで成り立っているわけではない。コア構造 (温度や気圧などのスカラー値、速度などのベクター値、圧力やゆがみなどのテンソル値) やレンダリング属性 (面法線やテクスチャ座標など) にも帰属する。

時空間情報を表すデータも、一般的にはサイエンティフィック・ビジュアライゼーションの一部だと考えられていることに注意しよう。しかし、もっと抽象的なデータ形式もある。マーケティングの統計や Web ページ、そしてドキュメントやその他情報で抽象的な関係 (構造化されていない文書や表、グラフ、ツリーなど) でしか表せないものなどだ。こういった抽象データは、一般的にはインフォメーション・ビジュアライゼーションで対応する。これらのコミュニティの助けを得て、VTK は今やサイエンティフィック・ビジュアライゼーションとインフォメーション・ビジュアライゼーションの両方に対応できるようになった。

ビジュアライゼーションシステムとしての VTK の役割は、これらの形式のデータを受け取って、最終的にそれを人間の五感で理解しやすい形式に変換することだ。したがって、VTK の重要な要件の一つは、データフローパイプラインを作ってデータの取り込みから処理、表現、レンダリングまでをできるようにすることになる。そのため、ツールキットは柔軟なシステムとして作られている必要があり、ツールキットの設計がいろんなレベルに影響する。たとえば、私たちは交換可能なコンポーネント群をまとめた形式で VTK を作ったが、これは意図的なものだ。このコンポーネントを組み合わせることで、さまざまなデータを処理できるようにした。

24.2　アーキテクチャの機能

VTK のアーキテクチャの詳細に深入りする前に、このシステムの開発と利用に大きな影響を与えた上位レベルの概念について説明する。そのひとつが、VTK のハイブリッドラッパー機能だ。この機能は、C++による VTK の実装から Python や Java そして Tcl などの言語バインディングを自動生成する (それ以外の言語に対応することも可能だし、追加されつつある)。開発の主力メンバーは C++で作業する。ユーザーやアプリケーション開発者も C++を使うことはあるだろうが、たいていは先にあげたようなインタプリタ型の言語のほうをより好む。コンパイル型とインタプリタ型の環境を組み合わせることで、それぞれのいいとこどりを目指した。ハイパフォーマンスコンピューティングを重視したアルゴリズムも使えるし、プロトタイピングやアプリケーション開発のときにも柔軟に対応できる。実際、この方式でいろんな言語での計算処理に対応できるようにしたおかげで、科学計算のコミュニティの多くの人たちに喜んでもらえた。彼らがソフトウェアを自作するときのテンプレートとしても VTK を使ってもらえることが多くなった。

ソフトウェア開発の面から見ると、VTK は CMake を使ってビルドを管理している。そして、テストには CDash/CTest を使い、クロスプラットフォームのデプロイに関しては CPack を使っている。実際、VTK はありとあらゆるプラットフォームでコンパイルできる。あきれるほど原始的な開発環境しかないことで有名なスーパーコンピュータでもだいじょうぶだ。さらに、開発ツールだけではなく Web ページや Wiki、(ユーザー向けおよび開発者向けの) メーリングリスト、ドキュメント生成機能 (Doxygen)、バグ追跡システム (Mantis) なども揃っている。

コア機能

VTK はオブジェクト指向のシステムなので、クラスやインスタンスのメンバーへのアクセスは VTK がきちんと管理している。すべてのデータメンバーは、protected あるいは private のどちらかになっている。これらにアクセスするには `Set` メソッドと `Get` メソッドを用い、Boolean データやモデルのデータ、文字列、そしてベクター用にそれぞれ特化したバージョンが用意されている。これらのメソッドの多くは、実際にはマクロとしてクラスのヘッダファイルに組み込まれる。つまり、たとえば

```
vtkSetMacro(Tolerance,double);
vtkGetMacro(Tolerance,double);
```

これを展開すると、次のようになる。

```
virtual void SetTolerance(double);
virtual double GetTolerance();
```

これらのマクロを使う理由は、単にコードをわかりやすくするというだけではない。それ以外にもいろんな理由がある。VTK には、デバッグを制御したりオブジェクトの変更時刻

(MTime) を更新したり参照カウンタを管理したりといった重要なデータメンバーが存在する。マクロを使えばこういったデータを正しく操作できるので、マクロを使うことが強く推奨される。たとえば、VTK のバグの中でも最もやっかいなものは、オブジェクトの MTime をきちんと管理していないときに発生するものだ。本来実行されるべきコードが実行されなくなったり、逆に必要以上に実行されてしまうことになる。

VTK の強みの一つが、データの表現や管理が比較的シンプルにできるという点だ。通常は、特定の型のデータ配列 (`vtkFloatArray` など) を使って連続する情報を表す。たとえば、三つの三次元座標のリストを表すには、九個のエントリ (x,y,z, x,y,z, など) を持つ `vtkFloatArray` を使う。これらの配列にはタプルの概念が存在するので、三次元座標は 3 タプルになるし、対称な 3×3 テンソルなら 6 タプルで表される (対称空間も保存できる)。この設計を選んだのは意図的なものだ。科学計算の世界では配列を操作するシステム (Fortran など) とのやりとりが一般的だし、メモリを確保したり解放したりするときにも連続データの大きなチャンクで扱うほうがずっと効率的だからである。さらに、通信やシリアライズそして IO といった操作は、一般に連続データのほうがずっと効率的になる。これらの (さまざまな型の) コアデータ配列が VTK のデータの大半を表す。便利なメソッドも多数用意されており、情報の追加や情報へのアクセスができるようになっている。高速アクセス用のメソッドもあるし、データの追加に合わせて必要に応じてメモリを自動的に確保するメソッドもある。データ配列は抽象クラス `vtkDataArray` のサブクラスである。これはつまり、汎用的な仮想メソッドを使ってシンプルにコーディングできるということだ。しかし、より高パフォーマンスを求める場合には、static なテンプレート関数を使う。これを使えば、型に応じて実装を切り替えて連続データの配列に直接アクセスできるようになる。

一般に、C++のテンプレートはクラスの公開 API からは見えないものだ。しかし、パフォーマンス上の理由でテンプレートは幅広く使われている。STL も同様だ。私たちは通常、PIMPL[1] パターンに従ってテンプレートの実装を隠蔽し、ユーザーやアプリケーション開発者からは見えないようにする。こうしておけば、先述のようにインタプリタ型言語のコードでラップすることになったとしてもうまくいく。テンプレートの複雑なところを公開 API からは見えないようにするということは、VTK の実装をアプリケーション開発者側から見たときにはデータ型を選ぶなんてことを気にせずに済むようになるということだ。もちろん、裏側では実際はデータ型を見て処理を切り替えているが、型の判定は実行時に実際にデータにアクセスするときに行っている。

VTK ではなぜ参照カウンタでメモリを管理しているんだろう？なぜ、ガベージコレクションのようなもっとユーザーにやさしい方法を使わないんだろう？そんなふうに思う人もいるだろう。手短に答えると、VTK ではデータの削除を完全に管理する必要があり、そのデータは巨大なサイズになる可能性があるからだ。たとえば 1000×1000×1000 バイトのデータのサイズはギガバイト単位になる。こんな巨大なデータを置きっぱなしにして、いつ解放するのかをガベージコレクタに判断させるというのは得策ではない。VTK では、ほとんどのクラス (`vtkObject` のサブクラス) が参照カウント機能を組み込みで持っている。すべてのオブジェ

[1] `http://en.wikipedia.org/wiki/Opaque\_pointer`

クトが参照カウンタを持っており、オブジェクトのインスタンスを作ったときにカウンタが1で初期化される。オブジェクトを登録して使うたびに、参照カウンタはひとつ増える。同様に、使っていたオブジェクトの登録を解除した場合 (あるいはオブジェクトを削除した場合) に、参照カウンタがひとつ減る。最終的にオブジェクトの参照カウンタがゼロになるのは、自分自身を破棄するときだ。一般的な例は、このようになる。

```
vtkCamera *camera = vtkCamera::New();   // 参照カウンタは 1
camera->Register(this);                 // 参照カウンタは 2
camera->Unregister(this);               // 参照カウンタは 1
renderer->SetActiveCamera(camera);      // 参照カウンタは 2
renderer->Delete();                     // レンダラを削除した後の参照カウンタは 1
camera->Delete();                       // カメラ自身の破棄
```

なぜそんなにも VTK にとって参照カウンタ方式が重要なのか。実はもう一つ別の理由がある。この方式のほうが、データを効率的にコピーできるからだ。たとえば、大量のデータ配列を含むデータオブジェクト D1 があるとしよう。この中には点やポリゴン、色、スカラー、テクスチャ座標などが含まれる。このデータを処理して新たなデータオブジェクト D2 を作ることを考える。このオブジェクトは、最初のデータオブジェクトに (点を指す) ベクターデータを追加したものだ。無駄の多いやりかたとしては、D1 を完全にディープコピーして D2 を作ってから、新しいベクターデータの配列を D2 に追加するという方法がある。それ以外には、まず空の D2 を作って D1 から D2 に配列をシャローコピーするという方法がある。参照カウンタを使えばデータの所有者をきちんと追えるので、あとは D2 に新たなベクター配列を追加するだけだ。後者の手法だと、データのコピーを回避できる。既に述べたとおり、これは優れたビジュアライゼーションシステムに不可欠なものだ。この章の後半で解説するデータ処理パイプラインは、この手の操作 (あるアルゴリズムへの入力から出力へとデータをコピーするという操作) をよく行う。そのため、VTK では参照カウンタが必須になる。

もちろん、参照カウント方式にもよく知られた問題はある。ときどき発生する循環参照がそのひとつだ。複数のオブジェクトがお互いに参照し合う状態になってしまうことがある。こんな場合には、きちんと考えた上での介入が必要になる。VTK の場合は、`vtkGarbageCollector` に組み込まれた特別な機能を使って循環参照状態のオブジェクトを管理する。循環参照のクラスが見つかると (開発中にはあり得ることだ)、そのクラスは自分自身をガベージコレクタに登録し、自身の `Register` メソッドと `UnRegister` メソッドをオーバーロードする。それ以降、オブジェクトの削除 (あるいは登録解除) メソッドは、ローカルの参照カウンタネットワークでトポロジー解析をして、互いに参照し合うオブジェクトの孤島を探す。そして、それをガベージコレクタが削除する。

VTK でのインスタンス作成の大半は、クラスの static メンバーとして実装されたオブジェクトファクトリー経由で行う。典型的な構文は、このようになる。

```
vtkLight *a = vtkLight::New();
```

このとき重要なのは、ここで実際にインスタンス化されるのは `vtkLight` ではないかもしれないということだ。`vtkLight` のサブクラス (`vtkOpenGLLight` など) になる可能性もある。

オブジェクトファクトリーを使う動機はいろいろあるが、中でも重要なのが、アプリケーションの可搬性を確保してデバイスに依存しなくできるということだ。たとえば先ほどの例では、レンダリングしたシーンの中にライトを作った。あるプラットフォーム上のあるアプリケーションでは `vtkLight::New` が OpenGL のライトになるかもしれないが、別のプラットフォームではそれ以外のレンダリングライブラリを使うことになるかもしれないし、そのグラフィックシステム上でライトを作る方法も違うかもしれない。実際にどの派生クラスのインスタンスを作るのかは、実行時のシステム情報に基づいて決める。初期の VTK には、gl や PHIGS、Starbase、XGL そして OpenGL など無数のオプションがあった。今ではこれらの大半が削除され、DirectX や GPU ベースなどの新たな方法が追加されている。VTK で書かれたアプリケーションは、新たなテクノロジーが登場しても書き換える必要がない。開発者は、新たなデバイスに対応した `vtkLight` のサブクラスやその他レンダリングクラスを作ってそのテクノロジーに対応するだけでいい。オブジェクトファクトリーを使う重要な利点がもう一つある。実行時に、パフォーマンス重視の別バージョンに置き換えられるということだ。たとえば `vtkImageFFT` は、特化型のハードウェアや数値計算ライブラリを使うものに置き換えることができる。

データ表現

VTK の強みのひとつが、複雑な形式のデータを表せることだ。シンプルな表形式から有限要素メッシュのような複雑な構造まで、幅広いデータを扱える。これらすべてのデータ形式は `vtkDataObject` のサブクラスになる。その様子を図 24.1 に示す (さまざまなデータオブジェクトクラスの一部だけを抜き出した継承ダイアグラムであることに注意)。

`vtkDataObject` の重要な特徴は、次のセクションで扱うビジュアライゼーションパイプラインで処理できるということだ。ここで示した数多くのクラスの中で、現実のアプリケーションで実際に使われるクラスは一部だけである。`vtkDataSet` とその派生クラス群はサイエンティフィック・ビジュアライゼーションに使う (図 24.2)。たとえば `vtkPolyData` は多角形メッシュ、`vtkUnstructuredGrid` はメッシュ、そして `vtkImageData` は二次元のピクセルや三次元のボクセルのデータを表す。

パイプラインのアーキテクチャ

VTK は、いくつかの主要なサブシステムで構成されている。ビジュアライゼーションパッケージと最も密接に関連するサブシステムは、データフロー/パイプラインのアーキテクチャだろう。概念的には、パイプラインアーキテクチャは三種類のオブジェクトで構成されている。ある形式のデータオブジェクトを処理したり変換したりフィルタしたりして別の形式にするオブジェクト (`vtkAlgorithm`)、そしてパイプラインを実行するオブジェクト (`vtkExecutive`) だ。パイプラインを実行するオブジェクトは、データオブジェクトとプロセスオブジェクト

図 24.1: データオブジェクトクラス

図 24.2: データセットクラス

を交互に連結したグラフ (パイプライン) を制御する。典型的なパイプラインの様子を図 24.3 に示す。

概念的には単純なものだが、パイプラインを実際に実装しようとするといろいろ大変だ。その理由のひとつが、データの表現方法が複雑に複雑になってしまう可能性があるというこ

図 24.3: 典型的なパイプライン

とである。たとえば、データセットの中には階層構造やグループ構造のデータを含むものもあるので、データを処理するときには、込み入った反復処理や再帰処理が必要になる。合成に関して言うと、並列処理をするには (共有メモリを使うにせよスケーラブルな分散方式を使うにせよ) データを分割しないといけない。このとき、分割した各部分が重なるようにしておかないと、一貫性のある境界情報 (導関数など) を計算できない。

アルゴリズムオブジェクトそれ自体も、独特の複雑さがある。アルゴリズムによっては、複数の入力を受け取ったり複数の出力を生成したりするものもあり、それらの型はすべて異なるかもしれない。ローカルのデータを処理するアルゴリズム (セルの中心を計算するなど) もあれば、グローバルな情報を必要とするアルゴリズム (ヒストグラムの計算など) もある。すべての場合において、アルゴリズムは自分への入力をイミュータブルなものとして扱わないといけない。つまり、アルゴリズムは、入力を読むだけで出力を生成しないといけない。ひとつのデータが複数のアルゴリズムへの入力になることもあり得るので、あるアルゴリズムが別のアルゴリズムへの入力をいじってしまうわけにはいかないからだ。

パイプラインの実行も、実行戦略によっては複雑になり得る。場合によっては、フィルタとフィルタの間の中間結果をキャッシュしたくなることもあるかもしれない。そうすれば、パイプラインが変更されたときに必要となる再計算の量を減らせる。一方、ビジュアライゼーションデータセットは巨大なものになる可能性がある。そんな場合は、その後の計算に不要となったデータはその場で解放したくなることだろう。さらに、パイプラインの実行にはいろいろ複雑な方法がある。たとえばデータを多重分解処理する場合などは、パイプラインを繰り返し処理しないといけない。

これらの考えかたのいくつかを示すため、そしてパイプラインのアーキテクチャについてさらに深く知るために、こんな C++の例を考えよう。

```
vtkPExodusIIReader *reader = vtkPExodusIIReader::New();
reader->SetFileName("exampleFile.exo");
```

```
vtkContourFilter *cont = vtkContourFilter::New();
cont->SetInputConnection(reader->GetOutputPort());
cont->SetNumberOfContours(1);
cont->SetValue(0, 200);

vtkQuadricDecimation *deci = vtkQuadricDecimation::New();
deci->SetInputConnection(cont->GetOutputPort());
deci->SetTargetReduction( 0.75 );

vtkXMLPolyDataWriter *writer = vtkXMLPolyDataWriter::New();
writer->SetInputConnection(deci->GetOuputPort());
writer->SetFileName("outputFile.vtp");
writer->Write();
```

この例では、リーダーオブジェクトが巨大な非構造化グリッド (あるいはメッシュ) のデータファイルを読み込む。次のフィルタは、メッシュから等値面を生成する。vtkQuadricDecimationフィルタは等値面のサイズを削減する。等値面は多角形のデータセットなので、その数を減らしていく (つまり、同じ高さの三角形の数を減らす)。最終的に、削減後の新しいデータファイルをディスクに書き出す。実際にパイプラインが実行されるのは、ライターが Write メソッドを実行したとき (データを書き込もうとしたとき) だ。

この例が示すように、VTK のパイプラインは要求駆動で実行される仕組みになっている。ライターやマッパー (データレンダリングオブジェクト) などの受信側でデータが必要になると、入力が欲しいというリクエストを出す。入力フィルタが既に適切なデータを持っている場合は、単に実行の制御を受信側に戻すだけだ。しかし、入力フィルタに適切なデータがない場合は、データを算出する必要がある。そのため、まずはデータがあるかどうかを入力フィルタにたずねることになる。このプロセスを上流に向けてさかのぼって、フィルタあるいはソースが「適切なデータ」を持っているところかあるいはパイプラインの開始地点に達するまで続ける。そこから適切な順番でのフィルタの実行が始まり、リクエストがあった時点までデータがパイプライン上を流れる。

「適切なデータ」とは何か、はっきりさせておこう。デフォルトでは、VTK のソースあるいはフィルタを実行すると、その出力をパイプラインがキャッシュすることになっている。将来無駄な再計算をせずに済ませるようにするためだ。このおかげで、計算や入出力によるメモリのコストを抑えられる。また、キャッシュの挙動は設定で変更できる。パイプラインがキャッシュするのはデータオブジェクトだけではない。データオブジェクトを生成したときの条件に関するメタデータもキャッシュの対象になる。このメタデータにはタイムスタンプ (ComputeTime) が含まれており、これがデータオブジェクトを計算した時刻を表す。つまり、最も単純に考えた場合の「適切なデータ」とは、パイプラインの上流にあるすべてのオブジェクトの更新時刻よりもあとで計算されたデータだということになる。この振る舞いを示すには、こんな例を考えてみればいい。先ほどの VTK の例の最後に、こんなコードを追加してみよう。

```
vtkXMLPolyDataWriter *writer2 = vtkXMLPolyDataWriter::New();
```

```
writer2->SetInputConnection(deci->GetOuputPort());
writer2->SetFileName("outputFile2.vtp");
writer2->Write();
```

先述のとおり、最初の writer->Write の呼び出しでパイプライン全体が実行される。writer2->Write() が呼ばれたときには、decimation フィルタがキャッシュした出力が最新であることをパイプラインが把握している。キャッシュのタイムスタンプを、decimation フィルタや contour フィルタそしてリーダーの最終更新時刻と比較すればわかることだ。したがって、このデータリクエストを過去の writer2 に渡す必要はない。では、コードをこんなふうに変更したらどうなるだろう。

```
cont->SetValue(0, 400);

vtkXMLPolyDataWriter *writer2 = vtkXMLPolyDataWriter::New();
writer2->SetInputConnection(deci->GetOuputPort());
writer2->SetFileName("outputFile2.vtp");
writer2->Write();
```

この場合は、前回 contor フィルタや decimation フィルタを実行した後で contour フィルタが変更されていることがわかる。したがって、これら二つのフィルタのキャッシュは無効となり、再実行しなければならなくなる。しかし、contour フィルタの前段階にあるリーダーには手が加えられていないので、このキャッシュは有効だ。リーダーは再実行する必要がない。

ここで示したシナリオは、要求駆動のパイプラインの中でも最もシンプルな例だ。VTK のパイプラインは、もっと洗練されている。フィルタや受信者がデータを必要とするときには、追加情報を指定して、特定のデータのサブセットだけを要求することができる。たとえば、フィルタがデータ片をストリーミングするようにして、out-of-core 分析を実行できる。先ほどの例をこのように変更してみよう。

```
vtkXMLPolyDataWriter *writer = vtkXMLPolyDataWriter::New();
writer->SetInputConnection(deci->GetOuputPort());
writer->SetNumberOfPieces(2);

writer->SetWritePiece(0);
writer->SetFileName("outputFile0.vtp");
writer->Write();

writer->SetWritePiece(1);
writer->SetFileName("outputFile1.vtp");
writer->Write();
```

ここでは、ライターがパイプラインの上流に対してデータを要求するときに、二つに分割して受け取るようにしている。それぞれを個別に読み込んで処理し、別々にパイプラインを流す。先ほど説明したような単純なロジックが、ここではうまくいかないことに気付いた人もいるかもしれない。あの単純なロジックだと、Write 関数が二度目に呼ばれたときにパイプラインを再実行しない。上流は何も変わっていないからだ。こんな複雑なケースに対応する

には、リクエストを処理するロジックにもう少し手を加えないといけないことになる。VTKにおけるパイプラインの実行は、複数のパスで構成されている。データオブジェクトの算出は、その中でも最後のパスになる。その前段にあるのが、リクエストパスだ。リクエストパスでは、受信者やフィルタから上流に対して、その後の計算結果の中で自分が欲しいものを伝えることができる。先ほどの例の場合、ライター側では、入力が欲しがっているのは2つあるデータのうちの0番のほうであることがわかる。このリクエストは、実際にはリーダーまでたらいまわしされる。パイプラインを実行すると、リーダーは、実際に必要なのがデータの一部であるということを知る。さらに、キャッシュされているデータのどの部分がオブジェクトのメタデータに格納されているのかという情報も知る。次にフィルタが入力からデータを要求されたときは、このメタデータをリクエストと比較する。この例の場合は、パイプラインを再実行して、別の部分を受け取るリクエストを処理する。

フィルタからできるリクエストには、それ以外にもいくつかの種類がある。時間ステップをリクエストしたり、構造化された範囲をリクエストしたり、ゴースト層(隣接情報を計算するときの、境界層)の数をリクエストしたりといったものだ。さらに、リクエストパスを処理する間に、各フィルタでリクエストに手を加えて下流に流すこともできる。たとえば、ストリームができないフィルタ(streamline フィルタなど)は、部分的なリクエストを無視してデータ全体を要求したりできる。

レンダリングサブシステム

ぱっと見た限り、VTKはシンプルなオブジェクト指向のレンダリングモデルを採用しており、3Dシーンを構成する各部品に対応するクラス群で構成されている。たとえば、vtkActorは vtkRenderer と vtkCamera でレンダリングするオブジェクトで、vtkRenderWindow の中には複数の vtkRenderer が存在することもある。シーンを彩るのは vtkLight だ。各 vtkActor の位置を制御するのが vtkTransform で、アクターの見た目を指定するには vtkProperty を使う。最後に、アクターの幾何学的表現を定義するのが vtkMapper だ。マッパーは VTK の中でも重要な役割を果たすものだ。データ処理パイプラインの最後をまとめ、レンダリングシステムへのインターフェイスとなる。こんな例を考えてみよう。データを削減して結果をファイルに書き出し、マッパーを使って結果を可視化してやりとりするものだ。

```
vtkOBJReader *reader = vtkOBJReader::New();
reader->SetFileName("exampleFile.obj");

vtkTriangleFilter *tri = vtkTriangleFilter::New();
tri->SetInputConnection(reader->GetOutputPort());

vtkQuadricDecimation *deci = vtkQuadricDecimation::New();
deci->SetInputConnection(tri->GetOutputPort());
deci->SetTargetReduction( 0.75 );

vtkPolyDataMapper *mapper = vtkPolyDataMapper::New();
```

```
mapper->SetInputConnection(deci->GetOutputPort());

vtkActor *actor = vtkActor::New();
actor->SetMapper(mapper);

vtkRenderer *renderer = vtkRenderer::New();
renderer->AddActor(actor);

vtkRenderWindow *renWin = vtkRenderWindow::New();
renWin->AddRenderer(renderer);

vtkRenderWindowInteractor *interactor = vtkRenderWindowInteractor::New();
interactor->SetRenderWindow(renWin);

renWin->Render();
```

この例では、アクターとレンダラーそしてレンダーウィンドウをひとつずつ作り、さらにマッパーを追加する。これは、パイプラインをレンダリングシステムにつなぐものだ。さらに、vtkRenderWindowInteractor を追加していることにも注目しよう。これはマウスとキーボードの入力をキャプチャするインスタンスで、それをカメラの操作などのアクションに変換する。この変換プロセスの定義は vtkInteractorStyle で行う (詳細は後述する)。デフォルトで、いろんなインスタンスやデータの値が裏側で設定される。たとえば ID の変換が行われるし、デフォルトのライト (ヘッドライト) がひとつ作られ、そのプロパティも設定される。

時を経て、このオブジェクトモデルもだんだん洗練されてきた。派生クラスを作ってレンダリングプロセスの特別な処理を開発するときなどは、少し複雑になる。vtkActor は vtkProp を特化したもの (つまり、ステージ上にあるプロパティ) であり、こういったプロパティを大量に使って、2D オーバーレイグラフィックスやテキスト、そして 3D オブジェクトのレンダリングを行う。さらには、ボリュームレンダリングや GPU の実装を使うなどの、より高度なレンダリングテクニックにも対応する (図 24.4 を参照)。

オブジェクトモデルと同様に、VTK がサポートするデータモデルも成長してきた。データとレンダリングシステムとの橋渡しをするマッパーにも、いろいろなものが出てきた。大きな拡張点は、変換の階層だ。当初は、単純にリニアな 4×4 の変換行列を使っていた。これが強力な階層構造になり、非リニアな変換にも対応するようになった。シンプレートスプライン変換などだ。たとえば、最初の vtkPolyDataMapper にはデバイスごとのサブクラス (vtkOpenGLPolyDataMapper など) があった。最近は、より洗練されたグラフィックスパイプライン (「ペインター」パイプライン) で置き換えられた。その様子を図 24.4 に示す。

ペインター方式は、さまざまなデータレンダリングテクニックに対応しており、それらを組み合わせて特殊なエフェクトを提供できる。この機能のおかげで、1994 年に実装されたシンプルな vtkPolyDataMapper をはるかにしのぐものになった。

さらにもうひとつ、ビジュアライゼーションシステムの重要な側面がある。選択サブシステムだ。VTK には「ピッカー」の階層がある。これは、ハードウェアベースの方法あるいはソフトウェアの方法 (レイキャスティングなど) にもとづいて vtkProp を選ぶオブジェクトを

図 24.4: ディスプレイクラス

おおざっぱにまとめたり、ピック操作の後のさまざまなレベルの情報を提供するオブジェクトをまとめたりする。たとえば、ピッカーの中には XYZ 空間での位置しか提供しないものもある。これは、どの `vtkProp` を選んでいるかは示さない。また、別のピッカーは、単に選ばれている `vtkProp` だけでなくそのプロパティのジオメトリを定義するメッシュを作る点あるいはセルを提供するものもある。

イベントとインタラクション

データを使ったインタラクションは、ビジュアライゼーションに不可欠なものだ。VTK では、いろんな方法によるインタラクションが発生する。いちばん単純なレベルだと、何かのイベントが発生したことに気付いたユーザーが何らかのコマンドを実行することができる (command/observer パターン)。`vtkObject` のすべてのサブクラスは、自分自身を登録している observer とそのオブジェクトのリストを保持している。登録のときに observer は、どのイベントを知りたいかを示す。さらに、そのイベントが発生したときに実行したい関連コマンドを指定することもある。その動きを見るために、こんな例を考えてみよう。あるフィルタ (ここでは polygon decimation フィルタ) がひとつの observer を持っており、この observer は `StartEvent` と `ProgressEvent` そして `EndEvent` の三つのイベントを監視している。これらのイベントが発生するのはそれぞれこんな場合だ。`StartEvent` はフィルタの実行が始まったと

き、と ProgressEvent 実行中に定期的に、そして EndEvent は実行完了時となる。次の例では vtkCommand クラスに Execute メソッドがあり、このアルゴリズムを実行するときに適切な関連情報を表示する。

```
class vtkProgressCommand : public vtkCommand
{
  public:
    static vtkProgressCommand *New() { return new vtkProgressCommand; }
    virtual void Execute(vtkObject *caller, unsigned long, void *callData)
    {
      double progress = *(static_cast<double*>(callData));
      std::cout << "Progress at " << progress<< std::endl;
    }
};

vtkCommand* pobserver = vtkProgressCommand::New();

vtkDecimatePro *deci = vtkDecimatePro::New();
deci->SetInputConnection( byu->GetOutputPort() );
deci->SetTargetReduction( 0.75 );
deci->AddObserver( vtkCommand::ProgressEvent, pobserver );
```

これは基本的なインタラクション形式だが、VTK を使う多くのアプリケーションで基本となるものだ。たとえば、先ほどのシンプルなコードに手を加えて、GUI のプログレスバーを表示させるのも簡単なことだ。この Command/Observer サブシステムは、VTK の 3D ウィジェットのでも重要な役割を果たしており、データの問い合わせや操作、編集などをよくできたインタラクションでできる。詳しくは後述する。

この例で重要なのは、VTK のイベントが事前に定義されているけれどもユーザー定義イベント用の裏口も用意されているという点だ。vtkCommand クラスが、イベントの列挙 (この例の場合は vtkCommand::ProgressEvent など) とユーザーイベントを定義している。UserEvent は単なる整数値で、一般的にはこれを開始オフセットとしてアプリケーション内でのユーザー定義イベントを扱う。つまり、たとえば vtkCommand::UserEvent+100 は、VTK で定義しているイベントとは別の何かのイベントを指すことになる。

ユーザーからは、VTK のウィジェットはシーンの中のアクターとして見える。ただ、通常のアクターとは違って、ハンドルやその他の幾何学的な機能を操作したりしてユーザーが操作できる (ハンドルの操作や幾何学的機能の操作は、先ほど説明したピッキング機能に基づいたものだ)。ウィジェットの操作は極めて直感的にできる。球状のハンドルをつかんで移動したり、直線をつかんで移動したりといったものだ。しかし、その裏側では、いろんなイベント (InteractionEvent など) を発行している。適切に書かれたアプリケーションはそのイベントを捕捉できるし、それにあわせて適切なアクションをとれる。たとえば、こんな感じで vtkCommand::InteractionEvent を引き起こすことがよくある。

```
vtkLW2Callback *myCallback = vtkLW2Callback::New();
  myCallback->PolyData = seeds;    // ストリームラインのシードポイントで、対話的に更新される
```

```
  myCallback->Actor = streamline;  // ストリームラインのアクターで、インタラクションで見え
るようになる

vtkLineWidget2 *lineWidget = vtkLineWidget2::New();
  lineWidget->SetInteractor(iren);
  lineWidget->SetRepresentation(rep);
  lineWidget->AddObserver(vtkCommand::InteractionEvent,myCallback);
```

VTK のウィジェットは、実際のところは二つのオブジェクトで構成されている。vtkInteractorObserver のサブクラスと vtkProp のサブクラスだ。vtkInteractorObserver は、単純にレンダーウィンドウ上でのユーザーのインタラクション (マウスやキーボードのイベントなど) を観察してそれを処理する。vtkProp のサブクラス (アクター) は、vtkInteractorObserver が操作する。一般に、この手の操作は、vtkProp のジオメトリを変更 (ハンドルを強調させるなど) したりカーソルの見た目を変更したり、データを変換したりといったものになる。もちろん、ウィジェットを操作するには、これらのサブクラスがウィジェットの振る舞いの細かい点を制御できるようにしておかないといけない。現時点で、システムには 50 以上のウィジェットが存在する。

ライブラリのまとめ

VTK は大規模なソフトウェアツールキットだ。現時点でのシステム全体のコードは、約 150 万行 (コメントを含む。ただし、ラッパーソフトウェアが自動生成したコードは含まない) であり、C++のクラス数は 1000 前後になる。こんな込み入ったシステムを管理するため、そしてビルドやリンクにかかる時間を減らすため、このシステムは多数のサブディレクトリで構成されている。表 24.1 にサブディレクトリの一覧をまとめた。各サブディレクトリのライブラリがどんな機能を提供するのかも簡単に示している。

24.3 ふりかえり/今後の展望

VTK は、これまでに大きな成功を収めてきた。最初の一行が書かれたのは 1993 年だが、この章の執筆時点でもまだ VTK は着々と成長を続けており、むしろ、どちらかといえば開発の速度は上がっている。[2] このセクションでは、これまでに学んだ教訓や今後の目標についてまとめよう。

プロジェクトの発展

VTK プロジェクトで最も驚くべきなのは、ここまで長い間生き延びてきたことだ。開発のペースは、これらの主要な要因に左右される。

[2] VTK の最新のコード分析が http://www.ohloh.net/p/vtk/analyses/latest で見られる。

Common	VTK コアクラス
Filtering	パイプラインデータフローの管理に使うクラス
Rendering	レンダリング、ピッキング、画像の表示、そしてインタラクション
VolumeRendering	ボリュームレンダリング
Graphics	3D ジオメトリ処理
GenericFiltering	ノンリニア 3D ジオメトリ処理
Imaging	画像パイプライン
Hybrid	グラフィック機能とイメージング機能の両方を必要とするクラス
Widgets	洗練されたインタラクション
IO	VTK の入出力
Infovis	情報の可視化
Parallel	並列処理 (コントローラとコミュニケータ)
Wrapping	Tcl や Python そして Java 用のラッパー
Examples	ドキュメントつきの豊富なサンプル

表 24.1: VTK のサブディレクトリ

- 日々追加され続ける、新たなアルゴリズムや新たな機能。たとえば、最近追加された新たな機能として、インフォマティクスサブシステムがある (Titan。主に Sandia National Labs と Kitware が開発している)。チャートやレンダリングのクラスも追加されたし、新たな科学データ型にも対応した。もうひとつ重要な追加機能が、3D インタラクションウィジェットだ。さらに、現在進行中なのが、GPU ベースのレンダリングやデータ処理機能で、これらによって VTK に新たな機能をもたらそうとしている。
- VTK の日々の露出や実際の利用例が増え続けるという自己永続的な流れによる、ユーザーや開発者の増加。たとえば、ParaView はサイエンティフィック・ビジュアライゼーションアプリケーションの中でも最も人気のあるものだが、これは VTK を使って書かれており、ハイパフォーマンスコンピューティングの世界で高い評価を得ている。3D Slicer は有名な生物医学コンピューティングプラットフォームだが、これも大半は VTK を使って書かれている。このプラットフォームは、年間何百万ドルもの資金援助を得ている。
- 日々成長し続ける VTK の開発プロセス。最近は、CMake や CDash、CTest、そして CPack などのツールが VTK のビルド環境に組み込まれた。さらに VTK のコードリポジトリは Git に移行し、より洗練されたワークフローが確立された。これらの改善のおかげで、VTK は今も科学コンピューティングの世界での開発の最先端にあり続ける。

これらの成長はすばらしいことで、このソフトウェアシステムを作ったのが間違いでなかったと感じるし、VTK の将来も有望だろう。でも、このながれをきちんと管理していくのはとても難しい。そこで、VTK の短期的な目標として、コミュニティの成長にもソフトウェアそ

のものの成長と同じように注力していくことにした。これは、次のような何段階かの手順で進めていった。

まず、きちんとした管理体制の構築を考えた。Architecture Review Board を作り、コミュニティや技術の発展を支援するようにした。ここでは、上位レベルの戦略的な課題に注力する。VTK コミュニティはさらに、Topic Lead のチームも編成した。このチームは、VTK の各サブシステムに関しての技術的な発展を支援する。

次に、ツールキットをさらにモジュール化するという計画を立てた。これは git のワークフロー機能を活用するものだが、ユーザーや開発者の要望に応えたものでもある。ユーザーや開発者はツールキット全体ではなく小さなサブシステムのほうが作業しやすいだろうし、パッケージ全体をビルドしたりリンクしたりするのも好まない。さらに、日々成長するコミュニティをサポートするために大切なのが、新機能やサブシステムへの貢献を受け入れることだ。たとえそれが必ずしもツールキットのコアには入らないものだとしても。疎結合なモジュール群による構成を作れば、周辺機能に関する大量の貢献を受け入れてもコアの安定性を保てる。

新技術

ソフトウェアの開発プロセスだけでなく、開発パイプラインの中ではいろんな技術的革新もあった。

- 共同処理機能は、ビジュアライゼーションエンジンをシミュレーションのコードに組み込み、定期的にビジュアライゼーション用データを生成する機能だ。この技術のおかげで、完全なソリューションデータを大量に出力しなくても済むようになった。
- VTK のデータ処理パイプラインは、まだまだ複雑すぎる。このサブシステムをリファクタリングでよりシンプルにする作業が進行中だ。
- データとの直接のインタラクション機能は、ユーザーに広く使われるようになってきた。VTK にはさまざまなウィジェット群が用意されているが、タッチスクリーンを使ったり 3D 方式を使ったりした新たなインタラクション技法も開発中だ。インタラクションの開発は急ピッチで進められている。
- 計算機化学は、材料デザイナーやエンジニアの間でその重要性を増している。そこで、化学データを可視化したり操作したりするための機能が VTK に追加された。
- VTK レンダリングシステムは、複雑すぎるとよく批判される。新たな派生クラスを作ったり新たなレンダリング技術に対応させたりするのが難しいというわけだ。さらに、VTK はシーングラフを直接サポートしていない。これもまた、多くのユーザーが望んでいるものだ。
- 最後に、新たなデータ形式も常に追加されている。たとえば医療の分野では、階層型の容積のデータセットをさまざまな解像度で対応している (共焦点顕微鏡法にローカルの拡大機能をあわせたものなど)。

オープンサイエンス

Kitware、もっと広い意味では VTK コミュニティは、オープンサイエンスにもコミットしている。これは、私たちがオープンデータやオープンパブリケーション、オープンソースなどに関わっていることを宣伝する方法のひとつだ。再現可能な科学システムを作っていることを保証するために、これらは不可欠だ。VTK は長年オープンソースかつオープンデータシステムとして配布されてきたが、ドキュメンテーションプロセスが欠けていた。よくできた書籍 [Kit10, SML06] はあったが、新たに貢献されたソースコードなども含む技術文書の収集は、アドホックな手段に頼っていた。この状況を改善するため、出版の仕組みも作った。*VTK Journal* [3] は、ドキュメントやソースコード、データ、テストイメージなどを記事として公開できるようにしたものだ。このジャーナルでは、コードの自動レビュー (VTK の品質保証テストプロセスを使ったもの) もできるし人間による投稿のレビューもできる。

教訓

VTK はこれまで無事に続いてきたが、うまくできなかったことだっていろいろある。

モジュラー設計: クラスのモジュール化に関しては、うまい判断ができた。たとえば、ピクセルごとにオブジェクトを作るなどというばかげたことはせず、より上位レベルの `vtkImageClass` を作って、その裏側でピクセルデータの配列を扱うようにした。しかし、中にはクラスを上位レベルでまとめすぎて複雑にしてしまったこともあった。結局はリファクタリングでより小さく分割することになったし、今もその作業は進んでいる。その最大の例が、データ処理パイプラインだ。最初は、パイプラインの実装は暗黙的なものだった。データオブジェクトとアルゴリズムオブジェクトのインタラクションとして実装されていたのだ。その後、パイプライン実行用のオブジェクトを明示的に作らないといけないことがわかった。このオブジェクトにデータとアルゴリズムのインタラクションを調整させ、いろんなデータ処理戦略を実装することになった。

重要な概念を使いこなせなかった: いちばん後悔していることのひとつが、C++のイテレータをうまく使いこなせなかったことだ。VTK では、データの走査の多くを科学計算用のプログラミング言語である Fortran 方式で行っている。イテレータの柔軟性をうまく活用できていれば、システムに大きな利益をもたらしていたことだろう。データの一部分だけを処理したり、条件を満たすデータだけを指定したりといったこともできたはずだ。

設計の問題: もちろん、設計上の判断の中にも、もっとうまくできただろうというものがたくさんある。データ実行パイプラインには手を焼かされ、何回もの作り直しを経て改善してきた。レンダリングシステムも複雑すぎて、それを派生した何かを作るのは難しい。もうひとつ難しいことがあって、これは VTK の当初の概念に由来するものだ。

[3] http://www.midasjournal.org/?journal=35

> 私たちはもともと、VTK を読み込み専用のビジュアライゼーションシステムと考えていた。データを見るためだけのシステムというつもりだったのだ。しかし、最近のユーザーはデータの編集もしたいと思うことが多い。データの編集を考えるなら、必要なデータ構造も大きく変わってくる。

VTK のようなオープンソースシステムのすばらしいところは、こういった多くの間違いを修正していけるという点だ。私たちには活発で有能な開発者コミュニティがついており、日々ソフトウェアの改良を続けている。将来もきっとこれが続くだろう。

第25章

Battle for Wesnoth

Richard Shimooka and David White

プログラミングって、単なる問題解決の方法だととらえられがちだ。要件を聞いた開発者が、それを解決すべくコードを書くっていう感じ。評価の基準は、技術的な実装の美しさとか効率とかそんなところになる。この本にも、そういった内容が満載だ。ただ、直接的な機能だけでなく、コードは人々の生活にも影響を及ぼすことがある。ここに参加して何か新しいものを作りたいという気持ちにさせることもできるんだ。残念ながら、プロジェクトに参加したいと考える人たちの前には、大きな壁が立ちはだかっている。

たいていのプログラミング言語は、いろいろな知識がないと使いこなせない。そこまでたどり着けない人も多いだろう。さらに、誰でも手を加えやすいコードを書くのは技術的に難しいし、別に手を加えやすくしなくたって、たいていのプログラムは書ける。ちょっとした書き捨てのスクリプトがきちんとしたきれいなプログラムに成長することなんか、ほとんどない。手を加えやすいコードにするには、プロジェクトやプログラムの設計に関して事前にしっかり考えないといけない。これは、ふつうにプログラミングを進めるときの直感に反することが多い。さらに、たいていのプロジェクトはある程度の実力があるメンバーに依存しており、十分に高いスキルを持っている。彼らにすれば、別に新たなメンバーが増えなくてもぜんぜん困らないんだ。その結果、他の人から見たコードのわかりやすさは軽視されるようになる。考慮するにしても、後付けで対応することになる。

Battle for Wesnoth プロジェクトでは、最初からこの問題に取り組んできた。これはターン制のファンタジー戦略ゲームで、GPL2 ライセンスのもとでオープンソースで公開されている。それなりに成功を収め、執筆時点では 4 百万ダウンロードをこえている。これはこれですごい数字だ。しかし、それよりも自慢したいことがある。いろんなスキルレベルの多数の人たちが、それぞれのやりかたで開発に参加するモデルだ。

コードに手を加えやすくするのは、開発者たちにとっては単なるあいまいな目標ではない。このプロジェクトの存亡に関わる重大なことだと考えられている。Wesnoth はオープンソースで開発されており、高いスキルを持つ開発者がたくさん集まってくれることはあまり期待できない。スキルレベルを問わずあらゆる人たちがプロジェクトにかかわれるようにしてお

くことを、長期的な生存戦略として考えた。

このプロジェクトでは、開発者たちが最初から、幅広い人たちが開発にかかわれるようにすることを試みてきた。それがプログラムのアーキテクチャ全体にわたって何らかの影響を及ぼしたことは間違いない。何らかの決断をするときは、たいていこのことを重視した。この章では、プログラムの内部について深く掘り下げ、どうやって手の加えやすさを向上させてきたかに注目して紹介する。

まずは、このプロジェクトでのプログラミングの概要を紹介する。使っている言語や依存関係、そしてアーキテクチャなどに関して扱う。次に、Wesnoth で使っている独自のデータストレージ言語である Wesnoth Markup Language (WML) について紹介する。WML の個別の機能について、特にゲーム内でのユニットへの効果を中心に説明する。それから、マルチプレイヤー機能の実装や外部プログラムについてとりあげる。最後に、プロジェクトの構造や参加者を増やすための取り組みについてまとめる。

25.1　プロジェクトの概要

Wesnoth のコアエンジンは C++で書かれている。執筆時点での行数は、ぜんぶで 200,000 行程度だ。これはゲームエンジンの中核となる部分で、コードベース全体からコンテンツを除いた半分程度になる。このプログラムでは、ゲームのコンテンツを独自のデータ言語で定義できるようになっている。その名は Wesnoth Markup Language (WML) だ。このゲームに同梱されている WML のコードは 250,000 行程度になる。プロジェクトの存在感が増すにつれて、コンテンツの割合が増えてきた。プログラムが成熟するにつれて、C++でハードコードされたゲームコンテンツも新たに書き直され、WML を使ってその動きを定義できるようになった。図 25.1 はプログラムのアーキテクチャの概要を示したもので、色が付いている部分は Wesnoth の開発者が保守するところ、そして白い部分は外部の依存関係を表す。

図 25.1: プログラムのアーキテクチャ

総合的には、このプロジェクトではできるだけ依存関係を減らすよう心がけている。アプリケーションのポータビリティを高めるためだ。こうしておけば、プログラムの複雑度がさ

がるという効果もある。さらに、開発者がいろんなサードパーティの API を学ぶ手間も省ける。もちろん、使うべきところでは適切な依存関係を利用する。それもまた同じ効果を与えてくれる。Wesnoth では、Simple Directmedia Layer (SDL) を使って画像処理や I/O そしてイベント処理を書いている。これを選んだ理由は、使いやすいことと多数のプラットフォームで共通の I/O インターフェイスが使えることだ。そのおかげで、プラットフォームごとに独自の API を書いていくのに比べてより幅広いプラットフォームへのポータビリティを確保できた。しかし、見返りとして失ったものもある。プラットフォームごとの独自機能を使うのが難しくなったのだ。SDL にはいろんなライブラリが添付されており、Wesnoth でもいろんなところでそれを使っている。

- SDL_Mixer(音楽や音声)
- SDL_Image(PNG などの画像フォーマットの読み込み)
- SDL_Net(ネットワーク I/O)

さらに、Wesnoth ではその他のライブラリも使っている。

- Boost(C++の高度な機能)
- Pango と Cairo(国際化フォント)
- zlib(圧縮)
- Python と Lua(スクリプトによる拡張への対応)
- GNU gettext(国際化)

Wesnoth エンジンでは、WML オブジェクト (文字列のディレクトリと子ノード群) をあらゆるところで使っている。WML ノードからさまざまなオブジェクトを構築したり、逆にオブジェクトを WML ノードにシリアライズしたりできる。エンジン内部ではこの WML ディレクトリベースの形式でデータを保持しているところもあり、C++のデータ構造を介さずに直接それをパースしている。

Wesnoth ではいくつかの重要なサブシステムを使っているが、その多くはできるだけ自己完結型になるようにしている。このように分割された作りにしておくことで、可読性をあげている。何か特定の部分に興味を持った人はすぐにその部分の作業にとりかかれるし、手を加えるときにも、それがプログラムの他の部分に悪影響を及ぼすことを心配せずに済む。主なサブシステムを次にまとめる。

- WML パーサーとプリプロセッサ
- 基本 I/O モジュール (ベースになる映像モジュールや音声モジュール、ネットワークモジュールのライブラリやシステムコールの抽象化)
- GUI モジュール (ボタンやリスト、メニューなどのウィジェットの実装を含む)
- ディスプレイモジュール (ゲーム盤やユニット、アニメーションなどのレンダリング)
- AI モジュール
- パスファインディングモジュール (六角形のゲーム盤を扱うさまざまなユーティリティ関数を含む)

- マップ生成モジュール (さまざまな形式のマップをランダムに生成する)

ゲームの流れを制御するモジュールも用意されている。

- タイトルスクリーンモジュール (タイトル画面の表示を制御する)
- ストーリーラインモジュール (一連のカットシーンを表示する)
- ロビーモジュール (マルチプレイヤーサーバー上でのゲームの設定画面を表示する)
- “play game” モジュール (ゲーム全体を制御するモジュール)

“play game” モジュールとディスプレイモジュールが、Wesnoth の中でも最大のものだ。その役割をはっきり定義することはできない。というのも、このモジュールの機能は常に変わり続けるものだし、明確な仕様を定めることなどできないからだ。そのため、このモジュールは常に、Blob アンチパターンに陥る危険にさらされている。いったい何をするものなのかわからない巨大な塊として君臨してしまう、というパターンだ。ディスプレイモジュールとこのモジュールのコードについては常にレビューされており、何か別のモジュールに切り出せるコードがないかどうかを調べている。

さらに、プロジェクト全体にかかわる補助機能もあって、これはメインプログラムとは切り離されている。マルチプレイヤーでのネットワークゲームを進行するマルチプレイヤーサーバー、そしてユーザーがコンテンツをアップロードして他のユーザーと共有できるようにするコンテンツサーバーなどがこれにあたる。どちらも C++で書かれている。

25.2 Wesnoth Markup Language

拡張可能なゲームエンジンである Wesnoth では、シンプルなデータ言語を使ってすべてのゲームデータの保存や読み込みをしている。XML を使うことも検討したけれど、結局は使わなかった。技術者じゃないユーザーにも使いやすいものにしたかったし、ビジュアルデータを使うときにも難しく考えずに済ませたかったからだ。そこで、独自のデータ言語を開発した。それが Wesnoth Markup Language (WML) だ。技術者じゃない普通の人たちが使うことを念頭に設計した。Python や HTML ですら尻込みしてしまうような人たちでも、WML なら何とかなるというふうにしたかったんだ。Wesnoth のゲームデータはすべて WML 形式で格納される。ユニットの定義、キャンペーン、シナリオ、GUI 定義、その他ゲームのロジックに関する設定も含めてすべてがこの形式になる。

WML の基本構造は XML と同じで、要素と属性で構成されている。しかし、要素の中にテキストを置くことはできない。WML の属性は、単に文字列と文字列をマップする辞書形式だ。そこに、属性を解釈するプログラムロジックを組み合わせたものになる。シンプルな WML の例を示す。これは、ゲームの中で使うエルフの戦士ユニットの定義を整えたものだ。

```
[unit_type]
    id=Elvish Fighter
    name= _ "Elvish Fighter"
```

```
    race=elf
    image="units/elves-wood/fighter.png"
    profile="portraits/elves/fighter.png"
    hitpoints=33
    movement_type=woodland
    movement=5
    experience=40
    level=1
    alignment=neutral
    advances_to=Elvish Captain,Elvish Hero
    cost=14
    usage=fighter
    {LESS_NIMBLE_ELF}
    [attack]
        name=sword
        description=_"sword"
        icon=attacks/sword-elven.png
        type=blade
        range=melee
        damage=5
        number=4
    [/attack]
[/unit_type]
```

Wesnoth では国際化を重視しているので、WML にも国際化機能が組み込まれている。属性の値の先頭にアンダースコアがついているものは、翻訳できるようになっている。翻訳可能な文字列の変換には GNU gettext を使う。WML をパースするときには、翻訳されたバージョンを利用する。

Wesnoth では、大量の WML ドキュメントを用意する方法ではなく一つのドキュメントにまとめてしまう方法を選んだ。ゲームエンジンの中に登場させるすべてのゲームデータを単一のドキュメントで管理することにしたのだ。これで、グローバル変数ひとつでドキュメントを維持できるようになった。たとえば、すべてのユニット定義を読み込むには、units 要素の中にある unit_type という要素を読めばいい。

すべてのデータを対年的に単一の WML ドキュメントに格納するとはいえ、それをすべて一つのファイルにまとめてしまうと扱いづらくなる。そこで、Wesnoth ではプリプロセッサをサポートすることにした。このプリプロセッサがすべての WML を処理してから、パースを行う。プリプロセッサを使えば、別のファイルの内容をインクルードしたり、特定のディレクトリの内容をすべてインクルードしたりもできるようになる。たとえば、

```
{gui/default/window/}
```

のように書くと、gui/default/window/の中にあるすべての.cfg ファイルをインクルードする。

WML はとても冗長になることがあるので、プリプロセッサではマクロも定義できるようにしている。マクロを使えば、冗長な部分を簡略化できる。たとえばエルフの戦士の定義の

中で{LESS_NIMBLE_ELF}を起動しているところは、マクロの呼び出しになっている。このマクロは、森の中にいるなどの特定の条件のもとで、敏捷性に劣るエルフの字ニットを作る。

```
#define LESS_NIMBLE_ELF
    [defense]
        forest=40
    [/defense]
#enddef
```

この設計の利点は、エンジン自体は WML ドキュメントがいくつのファイルに分かれているのかを知らなくて済むということだ。ゲームのデータをどのような構造にするのかを決めるのは WML の作者の役割となり、ファイルやディレクトリをどのように分けるのかも作者が決める。

ゲームエンジンが WML ドキュメントを読み込むときには、ゲームの各種設定に関連するプリプロセッサシンボルもいくつか定義する。たとえば、Wesnoth のキャンペーンではさまざまな難易度の設定を定義できる。このとき、それぞれの難易度設定が、別々のプリプロセッサシンボルとして定義される。たとえば、難易度を変える方法としてありがちなのが、敵に渡すリソース (ゴールド) の量を調整するというものだ。これを実現するには、こんな WML マクロを定義する。

```
#define GOLD EASY_AMOUNT NORMAL_AMOUNT HARD_AMOUNT
  #ifdef EASY
    gold={EASY_AMOUNT}
  #endif
  #ifdef NORMAL
    gold={NORMAL_AMOUNT}
  #endif
  #ifdef HARD
    gold={HARD_AMOUNT}
  #endif
#enddef
```

このマクロを、敵の定義の中でたとえば{GOLD 50 100 200} のようにして使う。これで、難易度レベルに応じて敵に渡すゴールドの量を変えられる。

WML の処理方法は条件によって変わるので、WML ドキュメントに渡したシンボルのどれかが Wesnoth エンジンの実行中に変化した場合は、WML ドキュメント全体を読み込み直してふたたび処理しないといけない。たとえば、ユーザーがゲームを開始したときに WML ドキュメントが読み込まれ、利用可能なキャンペーンやその他のデータが使えるようになる。しかしその後、ユーザーが何かのキャンペーンを開始して難易度レベルを (たとえば easy など) 指定したら、EASY を定義した上でドキュメント全体を読み直さないといけない。

この設計が便利なのは、単一のドキュメントにすべてのゲームデータをまとめられるという点と、シンボルを使って WML ドキュメントを簡単に設定できるという点だ。しかし、プロジェクトがうまく進むようになると、Wesnoth でもいろんなコンテンツが使えるようになってくる。その中にはダウンロード可能なコンテンツも含まれる。これらはすべて、最終的に

はコアドキュメントツリーに追加されるわけで、WML ドキュメントのサイズがメガバイト単位にふくれあがってしまうことになる。これはパフォーマンス的な問題の原因になる。ドキュメントを読み込むだけで 1 分ほどかかってしまうことになるかもしれないし、ゲームの途中でドキュメントの再読み込みが発生したときなどに遅延が生じてしまう。さらに、メモリも大量に消費することになる。その対策として、ちょっとした手を使っている。キャンペーンを読み込むときには、キャンペーンごとに一意なシンボルをプリプロセッサで定義する。そして、その作戦に固有のコンテンツは`#ifdef` し、必要なときだけ使うようにするのだ。

さらに、Wesnoth にはキャッシングシステムがあり、あらかじめ指定したキー定義群に関してプリプロセス済みの WML ドキュメントをキャッシュしている。もちろん、このキャッシングシステムでは WML ファイルのタイムスタンプをチェックしており、ファイルに変更があった場合はキャッシュを再生成する。

25.3 Wesnoth のユニット

Wesnoth の主役はユニットだ。エルフの戦士やエルフの女呪術師が、トロルの戦士やオークの兵卒と戦うことになる。基本的な挙動はすべてのユニットで共通だが、通常のゲームの流れを変えるような特別な能力もいろいろある。たとえばトロルはターンごとに自分の体を部分的に再生するし、エルフの女呪術師はエンタングリング・ルーツで敵を遅くさせることができ、ウーズは森の中では目に見えない。

これらをエンジンの中で表す、最適な方法は何だろうか。C++で基底クラス `unit` を作って、その派生クラスとしていろんなタイプのユニットを用意したくなるところだ。たとえば `unit` の派生クラスとして `wose_unit` クラスを作り、`unit` のほうでは仮想関数 `bool is_invisible() const` を用意して false を返すようにしておく。`wose_unit` ではそれをオーバーライドして、森の中にいるときは true を返すようにすればいい。

ルールが限られているときは、こんな手法でもうまくいくことだろう。ただ、残念ながら Wesnoth はとても大規模なゲームなので、そんなやりかたをしていれば拡張性を損ねてしまう。この方式の場合、もし新たなタイプのユニットを追加したくなったら新しく C++のクラスを作らないといけなくなる。さらに、別々の特性を組み合わせて使うことができない。再生能力があって、網で的のスピードを抑えられ、かつ森の中では見えなくなるというユニットがでてきたらどうする？それ用に新しくクラスを書かないといけなくなって、コードが重複してしまうことになる。

Wesnoth のユニットシステムは、継承を一切使わずにこの仕組みを実現した。`unit` クラスはユニットのインスタンスを表すものとして使い、それとは別に `unit_type` クラスを用意して、その型の全ユニットが共有する不変な性質を表した。`unit` クラスは、自分自身のオブジェクトの型への参照を持っている。考え得るすべての `unit_type` オブジェクトはグローバルな辞書で保持されており、メイン WML ドキュメントを読み込むときにこの辞書も読み込まれる。

ユニットタイプは、そのユニットが持つすべての能力のリストになる。たとえばトロルは“regeneration(再生)” 能力を持っていて、ターンごとに生命力を回復できる。トカゲの散兵には “skirmisher(散兵)” 能力があって、敵地を移動できる。これらの能力を認識する仕組みがエンジンに組み込まれている。たとえば開拓のアルゴリズムは、そのユニットに “skirmisher” フラグが設定されているかどうかを調べ、敵陣を自由に動けるかどうかを判断する。この手法だと、誰でも新しいユニットを追加できるようになる。エンジンが作った能力を自由に組み合わせるには、単に WML を編集するだけでいい。もちろん、まったく新しい能力やユニットの挙動を追加したりするにはエンジン自体を編集する必要がある。

さらに、Wesnoth の各ユニットには任意の数の攻撃方法を持たせることができる。たとえばエルフの射手は、長距離離用の弓による攻撃と短距離離用の剣による攻撃ができる。それぞれ、ダメージの量も違うし特徴も違う。こんな攻撃方法を表すために用意したのが `attack_type` クラスで、すべての `unit_type` のインスタンスは自分が使える `attack_types` のリストを持っている。

各ユニットにさらに個性を持たせるために、Wesnoth にはトレイトという機能もある。ユニットを補充するときに、ほとんどのユニットには定義済みトレイトからランダムに選ばれた二種類のトレイトが設定される。たとえば strong ユニットは格闘でより多くのダメージを与え、intelligent ユニットはレベルアップに要する経験値が減らせる。また、ゲームの中でユニットの装備を増やし、さらに強力にできる。たとえば、剣を手に取ったユニットは、攻撃時によりダメージを与えられるようになる。トレイトや装備の仕組みを実装するために、Wesnoth はユニットに手を加えられるようにした。WML で定義した変更を、ユニットのデータに適用する。この変更は、ある種の攻撃にも適用できる。たとえば strong トレイトは、格闘の際の攻撃のダメージは増やすが投射のときに使っても効果はない。

ユニットの挙動を WML だけで完全に設定可能にできればお見事というところだろう。じゃあ、いったいなぜ Wesnoth はそこにたどり着けていないのかを考えよう。もしユニットの挙動を好きにいじれるようにしようと思えば、WML はもっと柔軟にする必要があるだろう。データ指向の言語にするのではなく、もっと高機能なプログラミング言語にしてしまう必要がある。でも、そんなことをしたら、何か貢献しようと思っている多くの人たちが敷居を高く感じてしまうようになる。

さらに、Wesnoth の AI は C++で開発されており、ゲーム内のさまざまな能力を認識する。再生能力や見えなくなる力なども考慮してユニットの動きを決め、さまざまな能力を最大限に活用する。ユニットの能力を WML で定義することもできたが、今のように洗練された AI で能力を活用できるようにするのは難しくなっただろう。能力を使えるようにするけれども AI はそれを活用しないというのは、あまりうまい実装ではない。同様に、能力を WML で実装して、それを使うには AI の C++コードも修正しないといけないっていうのもうまくない。そこで、ユニットの定義は WML で行うけれども能力に関してはエンジンに組み込むという妥協案を採用した。これは Wesnoth の仕様の要件を十分満たす。

25.4　Wesnoth のマルチプレイヤー機能

Wesnoth のマルチプレイヤー機能の実装は、できるだけシンプルな手法を使うようにした。サーバーへの悪意のある攻撃の可能性をできるだけ減らすことには注意したが、チート対策にはそんなに力を入れていない。Wesnoth のゲーム中のあらゆる操作 (ユニットの移動、敵への攻撃、ユニットの追加など) はすべて、WML のノードとして保存される。たとえば、ユニットを移動させるコマンドはこんな WML ファイルになるだろう。

```
[move]
    x="11,11,10,9,8,7"
    y="6,7,7,8,8,9"
[/move]
```

これは、プレイヤーのコマンドに従ってユニットがたどる道を示している。このゲームには、このような WML のコマンドを受け取って実行する仕組みがある。これはとても有用だ。つまり、ゲームの初期状態とその後のすべてのコマンドを保存しておけば、そのゲームを完全に再現できるということだからだ。ゲームを再現できるようにしておけば、プレイヤーどうしがお互いのプレイを見るのにも便利だし、バグレポートのときにも便利に使える。

私たちコミュニティは、フレンドリーでカジュアルなゲームであることを心がけながら Wesnoth のネットワークマルチプレイヤー環境を作った。チート防止システムを組み込んで悪意のあるクラッカーと技術的に争うのではやめて、チート防止に無駄に力を入れすぎないようにした。私たちの競合だとみられている他のマルチプレイヤーゲーム (ランキングシステムで競争しあうもの) を分析して、クラッカー対策をどうするのかの参考にした。サーバー上のチート防止機能を意図的に外していると、チートしようとする人のやる気を大きくそぐようだった。さらに、モデレーターが前向きなコミュニティを作るようにし、個々の開発者とプレイヤーの信頼関係を築いて遊んでもらえるようにした。争うことよりもいい関係を保つことに注力したんだ。その努力の結果はうまくいっているものと見られる。悪意を持ってゲームをクラックしようという動きはほぼなくなったからだ。

Wesnoth のマルチプレイヤー実装は、ありがちなクライアント・サーバー型のインフラで構成されている。サーバー (wesnothd) は Wesnoth クライアントからの接続を受け付け、遊べるゲームの概要をクライアントに送信する。Wesnoth はプレイヤーに「ロビー」を表示し、どのゲームに参加するか、あるいは新たなゲームを作って他のユーザーを呼ぶのかを選べるようにする。プレイヤーが何かのゲームを選んでゲームが始まると、Wesnoth の各インスタンスはプレイヤーのアクションにしたがって WML コマンドを生成する。このコマンドはサーバーに送られ、サーバーは参加しているすべてのクライアントにそのコマンドを中継する。つまり、サーバーは、薄くシンプルな中継役として働くことになる。リプレイシステムを他のクライアントで使って、WML コマンドを実行する。Wesnoth はターン制のゲームなので、すべてのネットワーク通信には TCP/IP を使う。

このシステムでは、プレイヤー以外もオブザーバーとしてゲームを観戦できるようになっている。オブザーバーは進行中のゲームに途中参加できる。このとき、サーバーはゲームの

初期状態を表す WML を送信してから、それ以降に実行されたすべてのコマンドの履歴を送信する。これで、途中から参加したオブザーバーもゲームの状態についていけるようになる。この時点でゲームの履歴を見られるようになるが、オブザーバーがゲームの現在の状態に追いつくには多少時間がかかる。コマンドの履歴は高速に転送されるとはいえ、それでも少し時間がかかるからだ。もうひとつの方法としては、クライアントのひとつがゲームの現状のスナップショットを WML として生成して、それを新しいオブザーバーに送るというやりかたもある。しかし、この手法だとクライアント側にオブザーバーのオーバーヘッドを押しつけることになり、大量のオブザーバーを同時にゲームに参加させる DoS 攻撃もできるようになってしまう。

もちろん、Wesnoth クライアントはゲームの状態を何も共有せずただコマンドを送信するだけなので、クライアントがゲームのルールに合意することが重要になる。サーバーはバージョンによって分けられており、同じバージョンのゲームを使っているプレイヤーとしかやりとりできない。もし自分のクライアントのゲームのバージョンが他のクライアントとずれてしまったら、そのときにはプレイヤーにすぐ警告が出る。これは、チートを防ぐにも便利な仕組みだ。たとえクライアントをいじれば簡単にチートできるとしても、バージョンが変わればすぐに気づかれてしまうってことだ。

25.5 結論

Battle for Wesnoth のプログラムとしての長所は、幅広い人たちがコーディングに参加できるようになっているところだと思っている。この狙いを達成するため、あまりエレガントではないコードで妥協することも少なくない。ただ、プロジェクトに参加しているプログラマーの中でもデキる人たちはいて、そんな人たちは WML の構文が非効率的であることに不満を持っている。でも、ここでの妥協は、このプロジェクトの大成功の一因になった。今や Wesnoth は、ユーザーが作った何百ものキャンペーンや Era を誇る。これらの大半は、プログラミングの経験がほとんどない (あるいはまったくない) 人が作ったものだ。さらに、このプロジェクトがきっかけとなってプログラミングの仕事をするようになったという人も多い。彼らはこのプロジェクトでプログラミングを学んだんだ。これらが私たちのプロジェクトの目に見える成果で、ちょっと他のプログラムには類を見ないものだろう。

Wesnoth の歴史から学んで欲しい大切なことがひとつある。それは、経験の浅いプログラマーがどんな苦労をすることになるか、気をつけるということだ。実際にコードを書いて自分の力を伸ばしたいと思っている貢献者たちに立ちはだかる壁は何なのか、そこを気にしないといけない。たとえば、プログラムに対して何らかの貢献をしたいけれどもプログラミングに関する知識はまったくないという人もいるかもしれない。技術者向けのエディタ、たとえば `emacs` や `vim` の学習曲線は急勾配で、そんな人たちにとっては恐れ多いものだろう。そこで、WML はシンプルなテキストエディタでも扱えるような形式にした。誰でも貢献できるようなツールを用意したというわけだ。

しかし、コードを誰でも扱いやすくするというのは決して容易なことではない。「こうすればOK」というような明確なルールがないからだ。いろんな要素を比較した上でうまくバランスを取る必要がある。下手をすると、まったく逆効果になってしまうかもしれない。これがはっきりするのが、依存関係の扱いかた。依存関係が増えると参入障壁が高くなってしまうこともあるが、場合によっては依存関係のおかげで参入しやすくなることもある。あらゆる問題を、ケースバイケースで考えないといけないということだ。

Wesnothの成功の一部については、誇張しすぎないように注意が必要だ。このプロジェクトはかなり恵まれていたところもあって、そのあたりは他のプログラムではまねできないだろうからだ。コードを幅広い層に扱ってもらえるようにするためのひとつの要素は、そのプログラムの背景だ。Wesnothはオープンソースのプログラムだったので、この点ではいろいろ恵まれていた。より厳密に言うと、GNUライセンスのおかげで誰でも既存のファイルを見ることができたし、その動きを理解したり手を加えたりも自由にできたんだ。いろいろ試してみながら学んで、それを他の人たちと共有することが推奨される文化だが、これが他のプログラムにもうまくあてはまるとは限らない。とは言え、本章の中にはあらゆる開発者にとって有用な内容も含まれていると思う。自分たちが美しいコードを書いていく上での参考になれば幸いだ。

関連図書

[AF94] Rick Adams and Donnalyn Frey. *!%:: A Directory of Electronic Mail Addressing & Networks*. O'Reilly Media, Sebastopol, CA, fourth edition, 1994.

[Ald02] Gaudenz Alder. *The JGraph Swing Component*. PhD thesis, ETH Zurich, 2002.

[BCC+05] Louis Bavoil, Steve Callahan, Patricia Crossno, Juliana Freire, Carlos E. Scheidegger, Cláudio T. Silva, and Huy T. Vo. VisTrails: Enabling Interactive Multiple-View Visualizations. In *Proceedings of IEEE Visualization*, pages 135–142, 2005.

[Bro10] Frederick P. Brooks, Jr. *The Design of Design: Essays from a Computer Scientist*. Pearson Education, 2010.

[CDG+06] Fay Chang, Jeffrey Dean, Sanjary Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: a distributed storage system for structured data. In *Proc. 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI'06)*. USENIX Association, 2006.

[CIRT00] P. H. Carns, W. B. Ligon III, R. B. Ross, and R. Thakur. PVFS: A Parallel File System for Linux Clusters. *Proc. 4th Annual Linux Showcase and Conference*, pages 317âĂŞ–327, 2000.

[Com79] Douglas Comer. Ubiquitous B-Tree. *ACM Computing Surveys*, 11:121–137, June 1979.

[CRS+08] Brian F. Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. PNUTS: Yahoo!'s hosted data serving platform. *PVLDB*, 1(2):1277–1288, 2008.

[DG04] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Proc. Sixth Symposium on Operating System Design and Implementation*, 2004.

[DHJ+07] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's Highly Available Key-Value Store. In *SOSP'07: Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles*, pages 205–220, 2007.

[FKSS08] Juliana Freire, David Koop, Emanuele Santos, and Cláudio T. Silva. Provenance for Computational Tasks: A Survey. *Computing in Science and Engineering*, 10(3):11–21, 2008.

[FSC+06] Juliana Freire, Cláudio T. Silva, Steve Callahan, Emanuele Santos, Carlos E. Scheidegger, and Huy T. Vo. Managing Rapidly-Evolving Scientific Workflows. In *International Provenance and Annotation Workshop (IPAW)*, LNCS 4145, pages 10–18. Springer Verlag, 2006.

[GGL03] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google File System. *Proc. ACM Symposium on Operating Systems Principles*, pages 29âĂŞ–43, 2003.

[GL02] Seth Gilbert and Nancy Lynch. Brewer's Conjecture and the Feasibility of Consistent Available Partition-Tolerant Web Services. *ACM SIGACT News*, 33(2), 2002.

[GLPT76] Jim Gray, Raymond Lorie, Gianfranco Putzolu, and Irving Traiger. Granularity of Locks and Degrees of Consistency in a Shared Data Base. *Proc. 1st International Conference on Very Large Data Bases*, pages 365–394, 1976.

[GR09] Adam Goucher and Tim Riley, editors. *Beautiful Testing*. O'Reilly, 2009.

[Gra81] Jim Gray. The Transaction Concept: Virtues and Limitations. *Proc. Seventh International Conference on Very Large Data Bases*, pages 144–154, 1981.

[Hor05] Cay Horstmann. *Object-Oriented Design and Patterns*. Wiley, 2 edition, 2005.

[HR83] Theo Haerder and Andreas Reuter. Principles of Transaction-Oriented Database Recovery. *ACM Computing Surveys*, 15, December 1983.

[Kit10] Kitware. *VTK User's Guide*. Kitware, Inc., 11 edition, 2010.

[Knu74] Donald E. Knuth. Structured Programming with go to Statements. *ACM Computing Surveys*, 6(4), 1974.

[LA04] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proc. 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Mar 2004.

[LCWB+11] H. Andrés Lagar-Cavilla, Joseph A. Whitney, Roy Bryant, Philip Patchin, Michael Brudno, Eyal de Lara, Stephen M. Rumble, M. Satyanarayanan, and Adin Scannell. SnowFlock: Virtual Machine Cloning as a First-Class Cloud Primitive. *ACM Transactions on Computer Systems*, 19(1), 2011.

[Mac06] Matt Mackall. Towards a Better SCM: Revlog and Mercurial. 2006 Ottawa Linux Symposium, 2006.

[MQ09] Marshall Kirk McKusick and Sean Quinlan. GFS: Evolution on Fast-forward. *ACM Queue*, 7(7), 2009.

[PGL+05] Anna Persson, Henrik Gustavsson, Brian Lings, Björn Lundell, Anders Mattson, and Ulf Ärlig. OSS Tools in a Heterogeneous Environment for Embedded Systems Modelling: an Analysis of Adoptions of XMI. *SIGSOFT Software Engineering Notes*, 30(4), 2005.

[PPT+93] Rob Pike, Dave Presotto, Ken Thompson, Howard Trickey, and Phil Winterbottom. The Use of Name Spaces in Plan 9. *Operating Systems Review*, 27(2):72–76, 1993.

[Rad94] Sanjay Radia. Naming Policies in the Spring System. *Proc. 1st IEEE Workshop on Services in Distributed and Networked Environments*, pages 164–171, 1994.

[RP93] Sanjay Radia and Jan Pachl. The Per-Process View of Naming and Remote Execution. *IEEE Parallel and Distributed Technology*, 1(3):71–80, 1993.

[Shu05] Rose Shumba. Usability of Rational Rose and Visio in a Software Engineering Course. *SIGCSE Bulletin*, 37(2), 2005.

[Shv10] Konstantin V. Shvachko. HDFS Scalability: The limits to growth. *;login:*, 35(2), 2010.

[SML06] Will Schroeder, Ken Martin, and Bill Lorensen. *The Visualization Toolkit: An Object-Oriented Approach to 3D Graphics*. Kitware, Inc., 4 edition, 2006.

[SO92] Margo Seltzer and Michael Olson. LIBTP: Portable, Modular Transactions for UNIX. *Proc. 1992 Winter USENIX Conference*, pages 9–26, January 1992.

[Spi03] Diomidis Spinellis. On the Declarative Specification of Models. *IEEE Software*, 20(2), 2003.

[SVK+07] Carlos E. Scheidegger, Huy T. Vo, David Koop, Juliana Freire, and Cláudio T. Silva. Querying and Creating Visualizations by Analogy. *IEEE Transactions on Visualization and Computer Graphics*, 13(6):1560–1567, 2007.

[SY91] Margo Seltzer and Ozan Yigit. A New Hashing Package for UNIX. *Proc. 1991 Winter USENIX Conference*, pages 173–184, January 1991.

[Tan06] Audrey Tang. –Ofun: Optimizing for Fun. `http://www.slideshare.net/autang/ofun-optimizing-for-fun`, 2006.

[Top00] Kim Topley. *Core Swing: Advanced Programming*. Prentice-Hall, 2000.

これもきっと気に入ると思うんだ...

「このツールを使えばソフトウェア開発がより改善できるよ!」「このテクノロジーを使えば…!」「このプラクティスを使えば…!」…こんな主張があふれかえっている。しかし、その中で真実はどれくらいあるだろうか？中には、単なる希望的観測に過ぎないものもあるんじゃないかな？*Making Software* は、トップレベルの研究者や実務者たちが、ソフトウェア開発の世界での実証に基づくさまざまな発見をまとめたものだ。こんな疑問に対する答えが書かれている。

- 優秀なプログラマーは凡人の 10 倍の生産性がある？
- テストファーストを採用すれば、よりよいコードをより高速に書けるようになる？
- コードメトリクスで、ソフトウェアのバグの数を予想できる？
- デザインパターンって、実際のところどうなの？
- ペアプログラミングって、どんな効果がある？
- 地理的に離れていることと、組織体系上で離れていること。どちらのほうが影響が大きい？

The Architecture of Open Source Applications と同様、*Making Software* の収益もアムネスティ・インターナショナルに寄付される。

Making Software: What Really Works, and Why We Believe It
edited by Andy Oram and Greg Wilson
O'Reilly Media, 2010, 978-0596808327
`http://oreilly.com/catalog/9780596808303`

奥付

表紙の画像は Chris Denison の *48 Free Street Mural* in Portland, Maine からのもので、撮影者は Peter Dutton である。

表紙のフォントは Caroline Hadilaksono による Junction である。テキストのフォントは TeXGyre Termes で見出しのフォントは TeXGyre Heros、どちらも Bogusław Jackowski と Janusz M. Nowacki の作品だ。コードのフォントは Raph Levien による Inconsolata である。